A Scalable Galerkin Multigrid Method for Real-time Simulation of Deformable Objects

ZANGYUEYANG XIAN*, Shanghai Jiao Tong University and Microsoft Research Asia
XIN TONG, Microsoft Research Asia
TIANTIAN LIU, Microsoft Research Asia

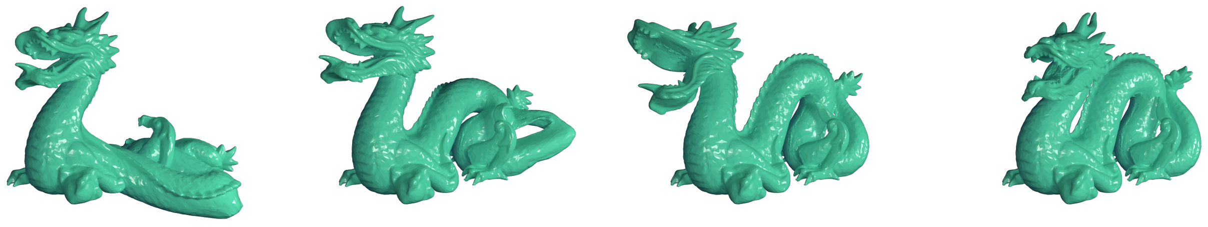

Fig. 1. Our multigrid method simulates a deformable dragon model with 200094 vertices and 676675 elements in its full space at 39.4 frames per second.

We propose a simple yet efficient multigrid scheme to simulate high-resolution deformable objects in their full spaces at interactive frame rates. The point of departure of our method is the Galerkin projection which is simple to construct. However, a naïve Galerkin multigrid does not scale well for large and irregular grids because it trades-off matrix sparsity for smaller sized linear systems which eventually stops improving the performance. Given that observation, we design our special projection criterion which is based on skinning space coordinates with piecewise constant weights, to make our Galerkin multigrid method scale for high-resolution meshes without suffering from dense linear solves. The usage of skinning space coordinates enables us to reduce the resolution of grids more aggressively, and our piecewise constant weights further ensure us to always deal with reasonably-sparse linear solves. Our projection matrices also help us to manage multi-level linear systems efficiently. Therefore, our method can be applied to different optimization schemes such as Newton's method and Projective Dynamics, pushing the resolution of a real-time simulation to orders of magnitudes higher. Our final GPU implementation outperforms the other state-of-the-art GPU deformable body simulators, enabling us to simulate large deformable objects with hundred thousands of degrees of freedom in real-time.

CCS Concepts: • **Computing methodologies** → **Physical simulation**.

Additional Key Words and Phrases: Physics-based animation, real-time simulation, multigrid.

*This work was done when Zangyueyang Xian was an undergraduate intern at Microsoft Research Asia.

Authors' addresses: Zangyueyang Xian, Shanghai Jiao Tong University and Microsoft Research Asia; Xin Tong, Microsoft Research Asia; Tiantian Liu, Microsoft Research Asia, ltt1598@gmail.com.

0730-0301/2019/11-ART162 $15.00
https://doi.org/10.1145/3355089.3356486

ACM Reference Format:
Zangyueyang Xian, Xin Tong, and Tiantian Liu. 2019. A Scalable Galerkin Multigrid Method for Real-time Simulation of Deformable Objects. *ACM Trans. Graph.* 38, 6, Article 162 (November 2019), 13 pages. https://doi.org/10.1145/3355089.3356486

1 INTRODUCTION

Deformable object simulation plays an important role in a variety of graphics applications to generate rich and vivid virtual contents. Many of these applications such as games, mixed reality, and surgical simulators require not only a visually acceptable deformation simulation, but more importantly, real-time interaction feedback. In order to provide stable large-time-step simulations of deformable objects in iterative frame rates, the implicit time integration scheme is usually considered as the standard way [Terzopoulos et al. 1987]. A classic numerical treatment to solve an implicit integration problem is Newton's method. However, Newton iterations are costly because of the frequent evaluation of state-dependent linear systems. Therefore, only low-resolution objects can be handled by Newton's method in real-time.

Fast numerical methods were investigated throughout the decades, enabling us to play with larger and larger meshes interactively. These methods usually departs from Newton's method with different strategies: they either approximate Newton directions using smart approximations in full space, such as Projective Dynamics [Bouaziz et al. 2014] and Position Based Dynamics [Müller et al. 2007], or simulate only a portion of degrees of freedom in subspaces which can be created using domain decomposition [Barbič and James 2005] or skinning spaces [Jacobson et al. 2012]. However, the performance of Projective Dynamics relies on pre-factorization of a direct solver, which can be hindered by high memory requirements for extremely high-resolution meshes. The convergence of Position Based Dynamics is not satisfying enough either, to simulate high-resolution and stiff objects, because of its usage of Jacobi or Gauss-Seidel iterations. Subspace methods scale well for high resolutions. But they are not able to handle the deformations that are not encoded in their subspaces.

Multigrid methods [McAdams et al. 2011; Tamstorf et al. 2015] are highly attractive when dealing with large linear system solves, since it scales very well with the system matrix size while spends typically less memory overhead compared to the original system matrix. A well designed multigrid method is able to simulate high-resolution objects in their full spaces while taking advantage of its multi-resolution hierarchies. However, constructing the multi-resolution data structure is often not an easy task to do.

We demonstrate an easy-to-construct yet scalable multigrid method to simulate high-resolution deformable objects in real-time. Our method starts from a Galerkin multigrid scheme [Strang and Aarikka 1986] which is known for its simplicity of set-up. However, a trivial set-up of a Galerkin multigrid does not scale for general unstructured meshes because of its high memory cost – especially in three-dimensional cases. We understand this side effect of Galerkin multigrid method from two aspects. One aspect is that the degrees of freedom of each coarser grid level cannot be decreased too aggressively from the finer grid, in order to maintain the convergence of a Galerkin multigrid. This would cause the size of coarse level systems larger than what we can afford. The other aspect is from the Galerkin projection itself – a smooth projection matrix can greatly increase the density of coarse grid linear systems, diminishing the returns from smaller linear solves. Our method uses the skinning space coordinates to represent the degrees of freedom for each vertex in coarse levels. We show that granting more information such as scaling and shearing to coarse level nodes is more efficient than increasing the coarse level resolution. Therefore, our method can reduce the size of each grid more aggressively compared to the original Galerkin multigrid. We further solve the dense matrix problem of a traditional Galerkin multigrid by setting piecewise constant interpolation weights. This set-up not only prevents our method from the curse of dense coarse level matrices, but enables us to efficiently update our coarse level linear systems according to the time-varying finest level linear system as well.

We deliver our multigrid solver as a general linear solve accelerator for deformable body simulations that does not assume a particular optimization scheme. Our method can be used to accelerate two-dimensional cloth and three-dimensional objects simulations using either Newton's method or Projective Dynamics. With the ability to support dynamic system matrices during a simulation, our method also handles all the run-time interactions such as dragging and collisions decently. As a result, we are able to simulate meshes with hundred thousands of degrees of freedom in real-time, as shown in Figure 1.

2 RELATED WORK

2.1 Time integration schemes

Deformable body simulation has gained its popularity since the pioneer work of Terzopoulos et al. [1987]. The key problem of deformable body simulation is to solve the equations of motion guided by Newtonian mechanics, updating the position and velocity of a deformable body using the external and internal elastic force. The most straightforward scheme to integrate the force is the explicit scheme, such as explicit Euler. However, for the purpose of animating deformable objects where performance prefers to use large time-steps, explicit methods are commonly not robust enough and suffer from numerical instability problems. Implicit schemes such as implicit Euler offer a robust simulation solution for large time-steps [Baraff and Witkin 1998], but they also rapidly remove the high-frequency motions from the simulation, reducing the system energy excessively. Symplectic integrators [Kharevych et al. 2006; Stern and Desbrun 2006] are well-known for their long term energy-conserving behaviors – the system simulated by a symplectic integrator will have its energy oscillate around the correct energy. But they are not guaranteed to be stable for all situations. Implicit Euler remains the most popular integrators for most of the graphics applications because of its stability. For the rest of this paper, we will also use implicit Euler as our discussed integrator.

2.2 Optimization integrators

Implicit Euler can be formulated as an optimization problem [Gast et al. 2015; Martin et al. 2011] which tries to find a compromise between a system's inertia and elastic energy. In order to solve this optimization problem, people usually rely on Newton's method as the core numerical solver because it converges fast. The cost of Newton's method is typically considered expensive since it requires to construct the system Hessian matrix, fix its definiteness [Smith et al. 2019; Teran et al. 2005] and solve the corresponding linear system in every Newton iteration. A clever quasi-Newton descent direction [Li et al. 2019] with careful line-search strategies [Zhu et al. 2018] can help to accelerate Newton's method, but is still not enough to meet the real-time requirement.

2.3 Fast approximate solutions

When real-time performance becomes a hard requirement in some applications, such as games, non-Newtonian methods are often used. Shape matching [Müller et al. 2005; Rivers and James 2007] reduces the degrees of freedom of an entire deformable body to some linear and higher order transformations, in order to achieve the real-time requirement. Inspired by shape matching methods, Position Based Dynamics(PBD)[Müller et al. 2007] and Nucleus solver [Stam 2009] provide more general solutions to simulate deformable bodies in real-time. PBD is one of the most popular techniques in real-time physically based simulation in the last decade and therefore is extended in many aspects. For example, better solver can be applied to deform large meshes [Müller 2008]; fluid is supported with PBD as well [Macklin and Müller 2013]; a unified multi-matter simulation behind NVIDIA Flex is also based on PBD [Macklin et al. 2014]. We refer to a great survey paper [Bender et al. 2015, 2014] for more details about PBD. The core projection step of PBD is very similar to a Gauss-Seidel or a Jacobi iteration of the Step and Project technique [Goldenthal et al. 2007] which treats the spring constraints as infinite stiff hard constraints. An eXtended version of PBD (X-PBD) [Macklin et al. 2016] applier the same strategy to a unified treatment of both soft and hard constraints using a complaint-constraint based framework [Tournier et al. 2015].

Another interpretation of PBD was observed by Liu et al.[2013], observing that PBD can be seen as an approximate integration of the implicit Euler method. This observation was further extended to a more general solver called Projective Dynamics (PD) [Bouaziz et al. 2014] that supports more soft constraints defined by finite elements method. Projective Dynamics uses a local/global update to solve the implicit Euler problem, where all the system nonlinearity is handled in the element-wise local steps, similarly to PBD. The global step solves a state-independent linear problem which can be pre-factorized using Cholesky factorization. Projective Dynamics can be interpreted as a Quasi-Newton method that approximate the time-dependent Hessian matrix using an example-dependent Laplacian matrix [Liu et al. 2017], therefore it can be generalized to support arbitrary hyper-elastic materials [Sifakis and Barbic 2012] and can be further accelerated using limited-memory BFGS update technique. Another interpretation of Projective Dynamics is a special case of Alternating Direction Method of Multipliers (ADMM) [Overby et al. 2017], leading to a different way to support more general materials. The key advantage of Projective Dynamics and its extensions is the pre-factorization of the system matrix, making the run-time linear solve requires only a forward/backward substitution. However, the substitution step is not parallel computing friendly. To make the most use of the power of GPU, iterative solvers such as Jacobi or Gauss-Seidel methods are preferred. Wang[2015] accelerated the iterative solvers for Projective Dynamics using a semi-iterative Chebyshev method and Fratacangeli et al.[2016] improves the parallelism of the Gauss-Seidel iteration using a novel coloring technique.

2.4 Reduced space methods

Unlike the full-space approximate methods, another way to cut the computation cost of simulating large deformable mesh is to reduce the simulation degrees of freedom using reduced spaces. Linear subspaces can be constructed using modal analysis methods, such as modal warping [Choi and Ko 2005] or modal derivatives [Barbič and James 2005]. Positional linear subspace can be constructed also using the skinning subspace [Jacobson et al. 2012; Wang et al. 2015]. Being not satisfied enough on the reduced linear system solve, hyper-reduced methods [Brandt et al. 2018; Von Tycowicz et al. 2013] also reduce the computation cost of the right-hand side of the linear system using carefully designed quadratures, pushing the performance of the reduced space methods to another level. Nonlinear subspace can also be constructed using autoencoder neural networks, restricting the system degrees of freedom in the latent space [Fulton et al. 2019]. The main problem of the reduced space methods is the lack of representing detailed deformations which were not accounted for during the subspace construction – a subspace method may not produce high-frequency motions if the subspace was constructed using a skinning space; example-based subspace methods may not work well under completely unseen deformations from the training data.

2.5 Multigrid methods

Multigrid methods offer an attractive solution where the simulation mesh is large and deformation details are rich [Georgii and Westermann 2006]. A proper multigrid method produces fine-scaled simulation details in full-space while accelerates the convergence from its multi-resolution subspaces. Finding proper interpolation and restriction criteria is the key to a successful multigrid framework. Cloth, as a two-dimensional model, is usually considered a good case for multigrid acceleration [Jeon et al. 2013; Tamstorf et al. 2015; Wang et al. 2018]. Structured three-dimensional multigrid is also a popular method because of its intuitive way to interpolate between the grid hierarchies [Dick et al. 2011; McAdams et al. 2011; Zhu et al. 2010]. Those regular grid structures are, by nature, consistent with the voxel representation, and can be also used to accelerate topology optimization problems [Liu et al. 2018; Wu et al. 2016]. However, a regular grid based algorithm is often difficult to apply to complex shapes or adaptively refined meshes. An efficient and simple-to-setup multigrid method for unstructured three-dimensional meshes remains a challenging problem. Our method is based on the Galerkin multigrid method, taking advantage of its simplicity to set up. We designed our Galerkin projection matrices to interpolate and restrict between different grid levels to simulate general deformable meshes including cloth and unstructured meshes.

3 BACKGROUND

3.1 Implicit time integration and descent methods

Deformable objects move in a virtual world with their own position $\mathbf{x}$ and velocity $\mathbf{v}$. At a certain time, a discrete version of the position and velocity are notated as $\mathbf{x}_n$ and $\mathbf{v}_n$. A time integration method can be seen as the art to predict the future state $\mathbf{x}_{n+1}$ and $\mathbf{v}_{n+1}$ with correct physics behavior. Implicit Euler method discretizes Newton's second law of motion as follows:

$$\mathbf{x}_{n+1} \quad = \quad \mathbf{x}_n + h\mathbf{v}_{n+1} \tag{1a}$$

$$\mathbf{v}_{n+1} \quad = \quad \mathbf{v}_n + h\mathbf{M}^{-1}\left(\mathbf{f}_{\text{int}} + \mathbf{f}_{\text{ext}}\left(\mathbf{x}_{n+1}\right)\right) \tag{1b}$$

where h is the time-step size, $\mathbf{M}$ is the mass matrix of, $\mathbf{f}_{\text{ext}}$ and $\mathbf{f}_{\text{int}}$ are the external and internal force respectively. For simplicity, damping is omitted in Eq. 1, and the internal force $\mathbf{f}_{int}$ is the elastic force which solely depends on the position $\mathbf{x}_{n+1}$. As hinted by Martin et al.[2011], we can substitute Eq. 1b into Eq. 1a and anti-differentiate the result equation on $\mathbf{x}$. This will give us the solution of the root finding problem Eq. 1 by minimizing an optimization problem:

$$g(\mathbf{x}) = \frac{1}{2h^2} ||\mathbf{x} - \mathbf{y}||_{\mathbf{M}}^2 + E(\mathbf{x}) - \mathbf{f}_{\text{ext}}^{\mathsf{T}}\mathbf{x} \tag{2}$$

where $||\cdot||_{\mathbf{A}}$ denotes the matrix norm $||\mathbf{x}||_{\mathbf{A}} = \sqrt{\mathbf{x}^{\mathsf{T}}\mathbf{A}\mathbf{x}}$; $E(\mathbf{x})$ is the elastic energy of the deformable object $\mathbf{f}_{int}(\mathbf{x}) = -\nabla_{\mathbf{x}}E(\mathbf{x})$; together with the negative sign, $-\mathbf{f}_{\text{ext}}^{\mathsf{T}}\mathbf{x}$ is the potential generated by the external force; and $\mathbf{y} = \mathbf{x}_n + h\mathbf{v}_n$ is an aggregated known vector the encodes the information of current position and velocity, indicating the future position as if there were no force. Once Eq. 2 is minimized, the solution of Eq. 1 is also found: $\mathbf{x}_{n+1} = \arg\min_{\mathbf{x}} g(\mathbf{x})$, where $\mathbf{x}_{n+1}$ balances the inertial term and the potential energy terms perfectly.

Due to the nonlinearity of $g(\mathbf{x})$, an analytical solution to minimize it is often not possible to evaluate. The common strategy is to iteratively descend the objective until finding a (local) minimum. A general descent method is described in Alg. 1, where the superscript $^{(k)}$ denotes the iteration count, and K is the maximum

ALGORITHM 1: General Descent Method.

```
1 initial guess: set $\mathbf{x}^{(0)}$.
2 for $k = 0, 1, \dots, K-1$ do
3     find descent direction: $\delta\mathbf{x}^{(k)}$;
4     decide step size: $\alpha > 0$;
5     commit descent direction: $\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + \alpha\delta\mathbf{x}^{(k)}$.
6 end
7 return $\mathbf{x}^{(K)}$
```

allowed number of iterations. The key of a descent method is to find the descent direction $\delta\mathbf{x}$. In order to compute $\delta\mathbf{x}$, the gradient information $\nabla g(\mathbf{x})$ is always involved. A common way to represent the descent direction is:

$$\delta\mathbf{x} = -\mathbf{A}^{-1}\nabla g(\mathbf{x}) \tag{3}$$

where the $\mathbf{A}$ matrix could be an arbitrary symmetric positive definite matrix. Different methods might prefer different choices of the $\mathbf{A}$ matrices and its linear solvers. Trade-offs between per-iteration performance and convergence speed are made in many real-time simulation algorithms. For example, Newton's method uses $\mathbf{A} = \nabla^2 g(\mathbf{x})$ to achieve nice convergence while Projective Dynamics approximates $\nabla^2 g(\mathbf{x})$ using a constant Laplacian-like matrix and pre-factorize the system matrix to achieve better per-iteration performance. A multigrid method could be the key to balance the convergence and the per-iteration cost in order to support a more efficient descent method.

3.2 Multigrid methods

A standalone multigrid method can be seen as one type of iterative methods to solve for linear systems. Stationary iterative methods such as Gauss-Seidel, Jacobi or their mixture successive over-relaxation, work very well when the system size is moderate. In higher resolution systems, they converge slower and might need to stop before converged. An early stop of these iterative methods produces a smooth error, which means the high-frequency error is removed quickly first, leaving only low-frequency error in the system. The multigrid methods offer a pleasant solution to mitigate this issue. The key observation of a multigrid method is that a low-frequency error in a finer level behaves like a high-frequency error in a coarser level. Under such observation, passing a smoother error from a finer grid level to a coarser level would make it "rougher", so that it can be removed in the coarse level iterations. Alg. 2 describes a simple two-grid v-cycle in a multigrid hierarchy. $\mathbf{R}$ is the restriction matrix that passes fine resolution values to coarse resolution; $\mathbf{P}$ is the prolongation (interpolation) matrix that interpolates coarse resolution values back to fine resolution; the coarse level solve $\mathbf{A}_2\mathbf{e}_2 = \mathbf{r}_2$ can be solved directly or iteratively, or even recursively by nesting another two-grid v-cycle into it. The $\mathbf{A}_2$ matrix in coarse level can be constructed using different ways. One way that naturally defines the $\mathbf{A}_2$ matrix is using the projection-style: $\mathbf{A}_2 = \mathbf{RAP}$. This is often referred as the Galerkin multigrid method.

Unlike pure geometry-based multigrid methods, Galerkin multigrid does not need to re-discretize the system matrix using finite element method on the coarser mesh. The restriction and interpolation

ALGORITHM 2: A two-grid v-cycle to solve $\mathbf{Ax} = \mathbf{b}$.

```
1 initial guess: set $\mathbf{x}_1$;
2 pre-smooth on $\mathbf{Ax} = \mathbf{b}$ to update $\mathbf{x}_1$;
3 compute residual $\mathbf{r}_1 := \mathbf{b} - \mathbf{Ax}_1$;
4 restrict the residual $\mathbf{r}_2 := \mathbf{Rr}_1$;
5 solve $\mathbf{A}_2\mathbf{e}_2 = \mathbf{r}_2$ to reach $\mathbf{e}_2$; (one might nest another two-grid
    v-cycle to solve this;)
6 interpolate the error $\mathbf{e}_1 = \mathbf{Pe}_2$;
7 update error to $\mathbf{x}_1$: $\mathbf{x}_1 := \mathbf{x}_1 + \mathbf{e}_1$;
8 post-smooth on $\mathbf{Ax} = \mathbf{b}$ to update the final $\mathbf{x}_1$;
9 return $\mathbf{x}_1$
```

steps are easy to set up, in fact, once the interpolation matrix $\mathbf{P}$ is decided, the restriction matrix is often set to $\mathbf{R} = \mathbf{P}^\mathsf{T}$ to make the coarse level system symmetric. The Galerkin projection also automatically handles the boundary conditions well, correctly passing the boundary conditions between different multigrid levels. Although holding all those sweet features, Galerkin multigrid method is seldom used in simulations of high-resolution deformable objects, especially for unstructured three-dimensional objects [Jacobson 2015]. We reason that as the bad memory consumption when stacking Galerkin multigrid v-cycles. The Galerkin projection matrices accumulate more and more non-zero values in coarser level system matrices which eventually stops it from gaining computational advantages on lower resolution grids.

Our method is also based on the Galerkin projection but scales for large unstructured meshes. We will show that by carefully designing the projection matrices, we can gain the multigrid acceleration on convergence without suffering from the curse of coarser level dense systems.

4 METHOD

In order to describe our Galerkin projection criterion, we first build our multi-level hierarchy by uniformly sampling vertices in multi-resolutions, we then set up the projection matrices using skinning-space coordinates with piecewise constant interpolation weights. We show that our projection matrices not only mitigate the memory cost issue for a Galerkin multigrid, but enables us to update our multi-level linear systems efficiently as well. There are edge cases where our projection matrices cause singularity, we will show our regularization approach to prevent undetermined solutions. To complete our multigrid method, we will also mention our choice of smoothers and bottom level solvers. At last, we explain how to use our method in a dynamics simulation with time-varying attachment and collision constraints.

4.1 Building the grid hierarchy

Assuming we are simulating a discretized deformable object with n vertices, let us denote $\Omega_0 = \{0, 1, \dots, n-1\}$ the set that contains the indices of all vertices on this full resolution deformable object. The positions of all the vertices are encoded in a single column vector $\mathbf{x} \in \mathbb{R}^{3n\times 1}$. One natural way to pick fewer degrees of freedom of a coarse level system is to find a subset of Ω_0: $\Omega_1 \subset \Omega_0$. In order to make the multigrid structure reasonable, we want the vertices in Ω_1 as representative as possible. Here we assume homogeneous materials, therefore a uniform sampling would make it work.

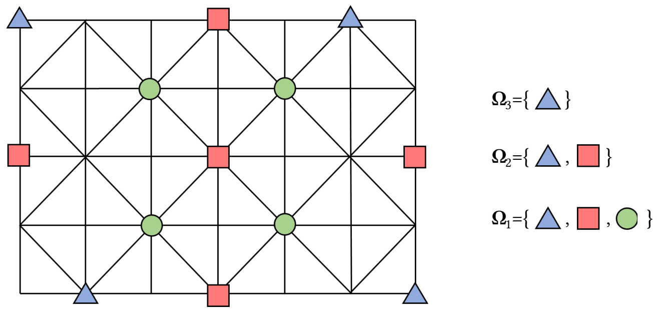

Fig. 2. Illustration of our grid hierarchy construction. Ω_l contains the vertices on the l-th level grid. Coarse level vertices form a subset of the fine level vertices: $\Omega_{l+1} \subset \Omega_l$.

Our approach follows the furthest point sampling method [Brandt et al. 2018] which can be seen as a special case of the k-means++ algorithm [Arthur and Vassilvitskii 2007]. Let us first consider a simple two-grid structure where Ω_0 contains all the full resolution vertices, and Ω_1 will be a subset of it with k_1 vertices ($k_1 < n$). We first initialize Ω_1 with a random vertex in Ω_0 and compute the geodesic distances to Ω_1 of all other vertices using Dijkstra's algorithm. We then pick the most distant vertex in $\Omega_0 \setminus \Omega_1$, add it to Ω_1 and update the geodesic distances to the new Ω_1. Note that the geodesic distance update is fairly lightweight because we only need to update around the most recently added vertex in Ω_1. We repeat this process until the size of Ω_1 is met. If multiple grid levels are needed, we simply pick the first k_{l+1} vertices from Ω_l as our Ω_{l+1}, $l = 1, 2, \ldots$. This automatically picks approximately equidistant vertices by the construction of Ω_l as we discussed above. Figure 2 illustrates this construction process. Note that although it is a bottom-up process, our grid construction does not always require the greedy algorithm to begin with. We can start with an arbitrary coarsest grid (e.g. picking vertices manually), and automatically grow the multigrid structure upon the initial grid.

4.2 Restriction and interpolation

Once the multi-resolution hierarchy is built, we need to decide the interpolation and restriction matrices **P** and **R** for the Galerkin projection. We first decide the degrees of freedom of coarse level variables. The degrees of freedom of a coarse grid is the product of two parts: the number of vertices of the grid and the degrees of freedom of each vertex on that grid. A trivial solution is to grant three degrees of freedom for each coarse level vertex because that fits the degrees of freedom of finest level grid data – the positions of full resolution vertices, making it easier and more consistent to interpolate data between levels [Wang et al. 2018]. However, we can not cut the degrees of freedom on coarser level grids too aggressively using this approach without hindering the convergence of a multigrid method. We found that increasing the degrees of freedom for each coarse level node improves our multigrid convergence more efficiently compared to increasing the number of coarse level vertices. (Our experiment can be found in Section 5.3.) Therefore, we use the skinning space coordinates [Brandt et al. 2018; Jacobson et al. 2012] where each coarse level node will have twelve degrees of freedom that can fully represent an affine transformation.

To use the skinning space coordinates, we start from linear blending skinning (LBS)[Magnenat-Thalmann et al. 1988] that computes the vertex positions as follows: $\mathbf{x}_i = \sum_j \omega_{ij} \mathbf{A}_j \mathbf{X}_i$, where $\mathbf{x}_i \in \mathbb{R}^{3\times 1}$ is the position of the i-th vertex, $\mathbf{A}_j \in \mathbb{R}^{3\times 4}$ is the affine transformation matrix of the j-th control handle, $\mathbf{X}_i \in \mathbb{R}^{4\times 1}$ is the homogeneous coordinate of the rest-pose position of vertex i, and ω_{ij} is the weight of handle j to vertex i which typically ranges from 0 to 1. Rearranging this LBS equation will give us a simple linear relation between the vertex positions and skinning space transformations:

$$\mathbf{x} = \mathbf{U}\mathbf{q} \tag{4}$$

where $\mathbf{x} = \left[\mathbf{x}_0^{\mathsf{T}}, \mathbf{x}_1^{\mathsf{T}}, \ldots, \mathbf{x}_{n-1}^{\mathsf{T}}\right]^{\mathsf{T}} \in \mathbb{R}^{3n\times 1}$ is the positions of all vertices; $\mathbf{q} = [vec(\mathbf{A}_0), vec(\mathbf{A}_1), \ldots, vec(\mathbf{A}_{k-1})]^{\mathsf{T}} \in \mathbb{R}^{12k\times 1}$ denotes all the skinning space degrees of freedom, notation $vec(\cdot)$ vectorizes a matrix to a row vector; and $\mathbf{U} \in \mathbb{R}^{3n\times 12k}$ is the linear transformation matrix between **q** and **x**, each block of **U** can be written down explicitly as $\mathbf{U}_{ij} = \omega_{ij}\mathbf{X}_i^{\mathsf{T}} \otimes \mathbf{I}_3 \in \mathbb{R}^{3\times 12}$ where $\otimes \mathbf{I}_3$ is a Kronecker product with a 3×3 identity matrix.

We pick **q** as our first level grid variable, so the interpolation matrix from this level to the finest level is automatically set: $\mathbf{P} = \mathbf{U}$. To ensure the symmetry of the system matrix in the first level using Galerkin-projection, we set the restriction matrix to $\mathbf{R} = \mathbf{U}^{\mathsf{T}}$. We keep using the skinning space coordinates as the degrees of freedom for coarser level grid variables, setting the interpolation laws as:

$$\mathbf{q}_l = \mathbf{U}_l \mathbf{q}_{l+1} \tag{5}$$

where $l = 1, 2, \ldots, \mathbf{q}_1 = \mathbf{q}$, and $\mathbf{U}_l$ is the linear interpolation matrix between the coarser level grids. Each block of $\mathbf{U}_l$ is simply a scaled 12×12 identity matrix: $\mathbf{U}_{l\,ij} = \omega_{l\,ij}\mathbf{I}_{12}$. Similarly, the coarser level restriction matrices are set as $\mathbf{R}_l = \mathbf{U}_l^{\mathsf{T}}$.

Now the only missing piece is the weight parameters, ω, that indicate how the coarse level variables will influence the finer level ones. The most intuitive way is to set up the weights as smooth as possible, for example, using bounded biharmonic skinning weights [Jacobson et al. 2011]. We tested this idea as our first attempt to set our weight parameters ω, and to further compute our interpolation matrix **U**. Under this setting, the multigrid method behaves great in terms of reducing the residual of a linear system. However, it works incredibly slow – we encountered exactly the same problem as we described in Section 3.2. In order to explain this, let us consider a two-grid structure, where two fine-grid vertices $\mathbf{x}_1$ and $\mathbf{x}_2$ are "affected" by n_1 and n_2 coarse-grid vertices respectively by some positive weights. Therefore an edge between $\mathbf{x}_1$ and $\mathbf{x}_2$ in the fine grid will produce $O((n_1 + n_2)^2)$ non-zeros in the coarse grid. Those fillings would make the coarse level system matrix $\mathbf{U}^{\mathsf{T}}\mathbf{A}\mathbf{U}$ much denser than it should be. Even worse, those dense data will be passed to the coarser levels, making their system matrices denser and denser. Figure 3 was produced from the octopus example with 39223 vertices. We pick 400 vertices to form a coarse level grid and analyze the coarse linear system. Although the degrees of freedom of the coarse grid is aggressively decreased from the full resolution

DoF 117669 to 4800 (keep in mind that each coarse grid vertex has 12 degrees of freedom), the number of non-zeros is only reduced roughly by a quarter if we use a smooth weighting strategy as shown in Figure 3(b).

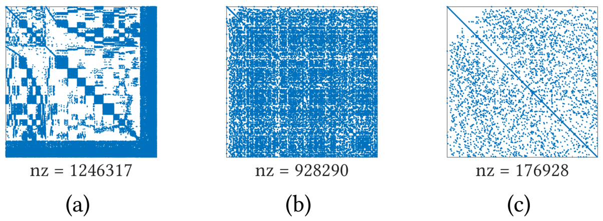

Fig. 3. The sparsity patterns and numbers of non-zeros. (a) the system matrix $\mathbf{A} \in \mathbb{R}^{117669\times117669}$ for the octopus example with 39223 vertices in Figure 11; (b) the coarse level grid matrix $\mathbf{U}^\mathsf{T}\mathbf{AU} \in \mathbb{R}^{4800\times4800}$ generated using bounded biharmonic weights; (c) the coarse level grid matrix $\mathbf{U}^\mathsf{T}\mathbf{AU} \in \mathbb{R}^{4800\times4800}$ generated by our piecewise constant weights. Both (b) and (c) pick the same 400 vertices as the coarse level grid.

To overcome the problem caused by dense systems in coarse levels, our method uses piecewise constant weight parameters: $\omega \in \{0, 1\}$. Each vertex in a finer grid is fully "controlled" by its closest vertex in the coarser grid, with the corresponding weight $\omega = 1$. Otherwise ω is set to zero. The intuitive part of this idea is to greatly reduce the number of non-zeros for coarse grids, which can be seen from Figure 3(c). However, this idea is usually considered counter-intuitive because discrete weights produce non-smooth results. On the contrary, we observe that the non-smooth results are not a big problem for multigrid methods because of the smoothing step. The artifact caused by non-smooth weight is often produced by an extremely high-frequency error – think of some vertices at the boundary between two "control handles", transforming the control handles separately will tear those boundary vertices apart, causing high-frequency errors which look like a cliff. Those high-frequency errors should be dealt with efficiently by stationary iterative solvers. In all of our test cases including cloth and three-dimensional mesh simulations, we observe that those high-frequency errors can be quickly removed with a few Gauss-Seidel or Jacobi iterations. Please refer to Section 5.2 for a more detailed analysis.

We only set up our interpolation and restriction matrices once before the simulation starts, and we will keep reusing the same $\mathbf{U}$ matrices during the simulation.

4.3 Fast update of multi-level system matrices

Another advantage of the 0-1 weights we used to construct our $\mathbf{U}$ matrices is the efficiency to update the system matrices for different level grids. The full resolution system matrix $\mathbf{A}$ may change all the time, so that the coarse level matrix $\mathbf{U}^\mathsf{T}\mathbf{AU}$ needs to update with $\mathbf{A}$ as well. For the first multigrid level, the system matrix can be rewritten as follows:

$$\mathbf{U}^\mathsf{T}\mathbf{AU} = \sum_{i,j} \mathbf{U}_i^\mathsf{T}\mathbf{A}_{ij}\mathbf{U}_j \tag{6}$$

where $\mathbf{A}_{ij} \in \mathbb{R}^{3\times3}$ is the i, j-th block of $\mathbf{A}$, $\mathbf{U}_i \in \mathbb{R}^{3\times12k}$ is the i-th 3×3 block-wise row of $\mathbf{U}$. From the definition of $\mathbf{U}$ as described after Eq. 4, in addition with our 0-1 weighting strategy, we know that the only non-zero block of $\mathbf{U}_i$ is $\mathbf{U}_{i,i_1} = \mathbf{X}_i^\mathsf{T} \otimes \mathbf{I}_3 \in \mathbb{R}^{3\times12}$, where i_1 is the index of the coarse level vertex that fully controls $\mathbf{x}_i$. After some numerical simplification, we can see that any update of $\mathbf{A}_{ij}$ will end up with a 12-by12 block update in $\left[\mathbf{U}^\mathsf{T}\mathbf{AU}\right]_{i_1,j_1}$:

$$\left[\mathbf{U}^\mathsf{T}\mathbf{AU}\right]_{i_1,j_1} = \mathbf{A}_{ij} \otimes \left(\mathbf{X}_i\mathbf{X}_j^\mathsf{T}\right) \tag{7}$$

where i_1 and j_1 are the indices of coarse level vertices that affects $\mathbf{x}_i$ and $\mathbf{x}_j$ respectively. Multiple different A_{ij} blocks can update to the same coarse level $\left[\mathbf{U}^\mathsf{T}\mathbf{AU}\right]_{i_1,j_1}$ when fine grid vertices i and j are affected by coarse level vertices i_1 and j_1, depending on the cross-grid weights. Even coarser level interpolation is trivial because the interpolation matrices between coarser levels are only made of 12×12 identity matrices as shown after Eq. 5.

Therefore, our method can also successfully handle the change of mass, time-step, or even rest-pose during the simulation as long as our $\mathbf{U}$ matrices are not changed.

4.4 Regularization for rank deficient cases

For our simulation cases, the system matrix $\mathbf{A}$ in Eq. 3 is guaranteed to be symmetric positive definite, then how about our coarse grid system matrices? According to Section 4.2, our method uses two types of interpolation matrices: $\mathbf{U} \in \mathbb{R}^{3n\times12k}$ for the first level grid and $\mathbf{U}_l \in \mathbb{R}^{12k_l\times12k_{l+1}}$, where k_l is the number of vertices we use for the l-th level grid, $k_1 = k$ and $k_{l+1} < k_l$. By construction of $\mathbf{A}_1 = \mathbf{U}^\mathsf{T}\mathbf{AU}$ and $\mathbf{A}_{l+1} = \mathbf{U}_l^\mathsf{T}\mathbf{A}_l\mathbf{U}_l$, we can see that our $\mathbf{A}_l$ matrices are symmetric positive semi-definite – they will be positive definite if they are not singular. It is easy to prove $\mathbf{A}_{l+1} = \mathbf{U}_l^\mathsf{T}\mathbf{A}_l\mathbf{U}_l$ is full-ranked as long as $\mathbf{A}_l$ is full-ranked: Our $\mathbf{U}_l$ is a Kronecker product between a 0-1 matrix and $\mathbf{I}_{12}$, that 0-1 weight matrix is full-ranked because each vertex in the coarser grid will at least fully control one vertex in the finer grid which is that vertex self, since $\Omega_{l+1} \subset \Omega_l$.

So let us focus on the first level grid matrix $\mathbf{A}_1 = \mathbf{U}^\mathsf{T}\mathbf{AU}$. Since $\mathbf{A}$ is positive definite, the only possibility to make $\mathbf{A}_1$ singular is $\mathbf{U}$ being rank-deficient: $rank(\mathbf{U}) < 12k$. In that case, the null space of $\mathbf{U}^\mathsf{T}\mathbf{AU}$ would be the same with $\mathbf{U}$. In theory, this is not a problem because the indeterminate solution of $\mathbf{U}^\mathsf{T}\mathbf{AUq} = \mathbf{U}^\mathsf{T}\mathbf{b}$ will be determinate after interpolated back to full space using $\mathbf{x} = \mathbf{Uq}$. However, we are not going to risk ourselves with dividing zero by zero cases because it is dangerous in practical implementations. The key idea is to regularize our $\mathbf{A}_1$ matrix only on its null space, so that we can stably solve for *one* solution of $\mathbf{U}^\mathsf{T}\mathbf{AUq} = \mathbf{U}^\mathsf{T}\mathbf{b}$.

Given the assumption that $\mathbf{A}$ being positive definite and $12k < 3n$, we know that $\mathbf{A}_1$ is only going to be singular if $\mathbf{U}$ is not full ranked. Ideally, each vertex in coarse level should contribute a rank-12 block to $\mathbf{U}$ to ensure the full rank of $\mathbf{U}$. However, that is not always the case. To explain this, let us take a deeper look of this block contributed by a single vertex in the course level, let us denote this block as $\mathbf{U}_{b3} = \mathbf{U}_b \otimes \mathbf{I}_3$, where

$$\mathbf{U}_b = \begin{bmatrix} \mathbf{X}_1^{(x)} & \mathbf{X}_1^{(y)} & \mathbf{X}_1^{(z)} & 1 \\ \mathbf{X}_2^{(x)} & \mathbf{X}_2^{(y)} & \mathbf{X}_2^{(z)} & 1 \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{X}_{n_b}^{(x)} & \mathbf{X}_{n_b}^{(y)} & \mathbf{X}_{n_b}^{(z)} & 1 \end{bmatrix} \tag{8}$$

For the simplicity of notation, we assume this coarse grid vertex controls n_b vertices whose indices ranges from 1 to n_b in the finest level. Let us also denote $\mathbf{A}_b$ as a sub-matrix of $\mathbf{A}$ that contains only the vertices indexed from 1 to n_b. $\mathbf{U}_{b3}^{\mathsf{T}}\mathbf{A}_b\mathbf{U}_{b3}$ shares exactly the same zero space with $\mathbf{U}_{b3}$ which is solely determined by $\mathbf{U}_b$. When would $rank(\mathbf{U}_b) < 4$? That would only happen when $\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_{n_b}$ lie in the same plane, or even worse, same line or same point. For example, in a cloth simulation where the cloth is initialized with a plane, $\mathbf{U}_b$ is always rank-deficient; in cases where one coarse grid vertex controls less than 4 fine level vertices, $\mathbf{U}_b$ will also be rank-deficient.

In order to regularize $\mathbf{A}_1$, we first compute the eigenvalues of $\mathbf{U}_b^{\mathsf{T}}\mathbf{U}_b \in \mathbb{R}^{4\times4}$ for each coarse grid vertex to detect if any of them is close to zero. We then find the corresponding eigenvector $\mathbf{n}$. We denote this eigenvector as $\mathbf{n}$ because it can be interpreted geometrically as the normal direction of the degenerated space of $\{\mathbf{X}_1, \cdots, \mathbf{X}_{n_b}\}$. And we finally add $\mathbf{n}\mathbf{n}^{\mathsf{T}} \otimes I_3$ to $\mathbf{U}_{b3}^{\mathsf{T}}\mathbf{A}_b\mathbf{U}_{b3}$ to regularize the whole system where the $\mathbf{n}$ vector simply "lifts up" that degenerated space back to a three dimensional one. Note that we do not need any scaling parameters before $\mathbf{n}\mathbf{n}^{\mathsf{T}}$. Because no matter how much we regularize the system, the regularization term only changes the values that lie in the null space of $\mathbf{U}$ which will vanish after we interpolate them back using $\mathbf{x} = \mathbf{U}\mathbf{q}$. This regularization only depends on $\mathbf{U}$ matrix, therefore we only need to handle it once in our pre-computation step.

4.5 Smoother and coarsest level solver

In theory, we can adopt any stationary iterative solvers that produce smooth errors as our smoother. Remember that our linear system matrices are stored in blocks where $\mathbf{A}$ is stored using 3×3 blocks and the coarse level matrices $\mathbf{A}_l$ are stored using 12×12 blocks, we use block iterative solvers with corresponding block sizes to smooth our linear systems. In practice, we apply block Jacobi iterations for mass-spring systems behind our cloth simulations because of its pleasing per-iteration performance. For three-dimensional finite element methods, we use a colored symmetric block Gauss-Seidel method [Fratarcangeli et al. 2016] as our smoother, since the system matrices in FEM simulations are no longer diagonally dominant, hurting the convergence for Jacobi iterations without over-relaxation.

Our coarsest level grid size is typically small, ranging from tens to hundreds of degrees of freedom. Therefore, we choose to solve our coarsest level linear system using dense Cholesky factorization technique.

4.6 Dynamics

Our method is used to solve for Eq. 3 in dynamics simulations of deformable objects. As we do not rely on any pre-factorization, our method does not prefer any specific choices of the system matrix $\mathbf{A}$. The only assumption we made is that $\mathbf{A}$ being symmetric positive semi-definite so that our stationary iterative solvers and dense Cholesky factorization will work. Note that this assumption also guarantees $\delta\mathbf{x}$ to be a descent direction, which makes the whole descent method (Alg. 1) work. Online update of the system matrix $\mathbf{A}$ due to time-varying attachment or collision constraints is naturally supported in our multigrid method. Thanks to our multigrid structure, we are able to quickly update the multi-level matrices to account for the online update of $\mathbf{A}$ using Eq. 7.

Attachments. We use soft attachment constraints to pin an object to a certain place, or to drag some vertices around due to user interaction. The attachment constraint is similar to connect a mesh vertex to some target position using a zero-length spring. The energy of an attachment constraint can be seen as: $E_{\text{att}}(\mathbf{x}_{\text{att}}) = \frac{1}{2}k_{\text{att}}\,||\mathbf{x}_{\text{att}} - \mathbf{t}_{\text{att}}||^2$, where k_{att} is the attachment stiffness, $\mathbf{x}_{\text{att}}$ is the vertex to be attached and $\mathbf{t}_{\text{att}} \in \mathbb{R}^{3\times1}$ is the target position. The second order derivative $\nabla^2_{\mathbf{x}_{\text{att}}}E_{\text{att}} = k_{\text{att}} \otimes I_3$ can be dynamically added or removed from system matrix in Eq. 3, enabling users to move vertices interactively.

Collisions. A similar strategy is applied to handle collisions, we detect inter-penetrations at the beginning of every iteration, and attempt to move the collided vertex out of its collision surface using a quadratic penalty energy: $E_{\text{col}}(\mathbf{x}_{\text{col}}) = \frac{1}{2}k_{\text{col}}\left((\mathbf{x}_{\text{col}} - \mathbf{t}_{\text{col}})^{\mathsf{T}}\,\mathbf{n}\right)^2$, where k_{col} is the collision stiffness, $\mathbf{x}_{\text{col}}$ is the collided vertex, $\mathbf{t}_{\text{col}}$ is the closest surface vertex and $\mathbf{n} \in \mathbb{R}^{3\times1}$ is the normal direction of the collided surface. This energy is only enabled when the vertex is penetrating the collision surface, i.e. $(\mathbf{x}_{\text{col}} - \mathbf{t}_{\text{col}})^{\mathsf{T}}\,\mathbf{n} < 0$, otherwise the collision energy is set to zero. The contribution to the system matrix due to one collision can be written as $\nabla^2_{\mathbf{x}_{\text{col}}}E_{\text{col}} = k_{\text{col}}\mathbf{n}\mathbf{n}^{\mathsf{T}} \in \mathbb{R}^{3\times3}$. Unlike Projective-Dynamics-based methods which are often reluctant to update this anisotropic collision Hessian to the system matrix [Liu et al. 2017], our method correctly accounts for the collision when computing the system matrix. Some results of our simulations with collisions can be seen in Figure 4.

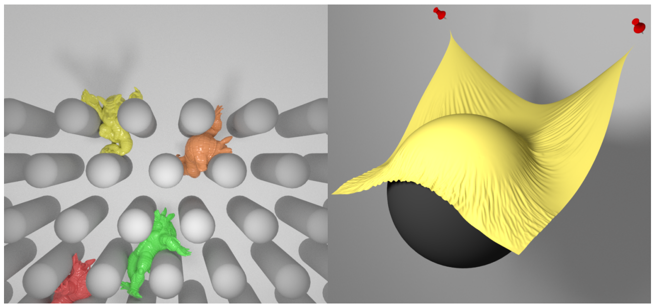
Fig. 4. Dropping armadillos into a Pachinko machine (left), with only static collisions; Draping a piece of cloth over a sphere (right).

4.7 Pipeline summary

To put all the pieces together, we summarize our simulation pipeline in Alg. 3 and Alg. 4, where the initialization step is only executed once before the entire simulation, and the update step is computed every frame.

5 RESULTS

5.1 Settings and performance

Table 1 reports all the testing scenarios we used in this paper and in our accompanying video. All examples are executed on an Intel Xeon 3.7 GHz CPU and an NVIDIA GeForce RTX 2080 GPU. Most of the run-time algorithms are implemented in GPU using

ALGORITHM 3: Initialization

```
1  set the max depth $d$ of a multigrid: $d >= 2$;
2  run furthest point sampling to construct $\Omega_0, \ldots, \Omega_{d-1}$; (Section 4.1)
3  construct our projection matrices $\mathbf{U}_0, \ldots, \mathbf{U}_{d-1}$; (Section 4.2)
4  find rank-deficient subspaces and their corresponding degenerated
     eigendirections $\mathbf{n}$. (Section 4.4)
5  if running Newton's method then
6      analyze sparsity pattern for the system Hessian $\mathbf{A}_0$;
7      analyze sparsity pattern for multi-level system matrices
         $\mathbf{A}_1, \ldots, \mathbf{A}_{d-1}$;
8      allocate memory for multi-level system matrices $\mathbf{A}_0, \ldots, \mathbf{A}_{d-1}$.
9  else
10     compute and store the system matrix $\mathbf{A}_0 := \mathbf{M}/h^2 + \mathbf{L}$; ([Bouaziz
         et al. 2014])
11     compute and store the multi-level system matrices
         $\mathbf{A}_1, \ldots, \mathbf{A}_{d-1}$.
12 end
```

ALGORITHM 4: Update

```
1  handle user interactions, generate run-time attachment constraints;
2  initialize $\mathbf{x} := \mathbf{x}_n + h\mathbf{v}_n$.
3  while max iteration count not reached & not converged do
4      run collision detection, generate collision constraints;
5      if running Newton's method then
6          fill the definiteness-fixed system Hessian matrix
             $\mathbf{A}_0 := \mathbf{M}/h^2 + \nabla^2_{\mathbf{x}} E(\mathbf{x})$;
7          fill the multi-level matrices $\mathbf{A}_1, \ldots, \mathbf{A}_{d-1}$.
8      end
9      regularize $\mathbf{A}_1$ using $\mathbf{n}$ if necessary; (Section 4.4)
10     update attachment and collision constraints to $\mathbf{A}_0, \ldots, \mathbf{A}_{d-1}$;
         (Section 4.3)
11     compute gradient $\nabla_{\mathbf{x}} g(\mathbf{x})$;
12     find descent direction $\delta\mathbf{x} := -\mathbf{A}_0^{-1}\nabla_{\mathbf{x}} g(\mathbf{x})$ using our multigrid
         solver; (Alg. 2)
13     decide step size $\alpha > 0$ using line search;
14     commit descent direction $\mathbf{x} := \mathbf{x} + \alpha\delta\mathbf{x}$.
15 end
16 update position: $\mathbf{x}_{n+1} := \mathbf{x}$;
17 update velocity: $\mathbf{v}_{n+1} := (\mathbf{x}_{n+1} - \mathbf{x}_n)/h$.
```

CUDA library, pre-computations such as setting up the grid hierarchy and compute the $\mathbf{U}$ matrices are mostly handled on CPU. For all experiments, we use implicit Euler with a fixed 1/30s time-step as our time integrator. The material properties of our simulated examples can be seen in the appendix. We run Projective Dynamics [Bouaziz et al. 2014] on our three-dimensional finite element simulations because we find the result visually plausible enough for most cases. In cases where vivid high-frequency effects take place, like a cloth simulation, we use Newton's method to compute the descent directions. A definiteness fix is needed when running Newton's method where the Hessian matrix is not guaranteed to be positive definite. In these cases, we follow the method in [Teran et al. 2005] to project the element-wise Hessian blocks into positive semi-definite matrices before assembling them into the final Hessian matrix. We keep nesting the two-grid c-cycles to form a V-cycle in our experiments. The number of grid levels, the number of vertices in each grid level and the choice of the linear solver for each level are reported in Table 1. For example, in the fourth row of the armadillo example in Table 1 with a grid set-up "25/500/all" and a solver set-up "direct/GS(3)/GS(3)", we use three grid levels: the coarsest level grid contains 25 vertices and is solved by a direct solver using dense Cholesky factorization; the middle level grid contains 500 vertices and is pre- and post-smoothed by 3 iterations of Gauss-Seidel respectively; the finest level grid contains all the mesh vertices and is also pre- and post-smoothed by 3 Gauss-Seidel iterations. All our examples are simulated in real-time with more than 30 frames per second.

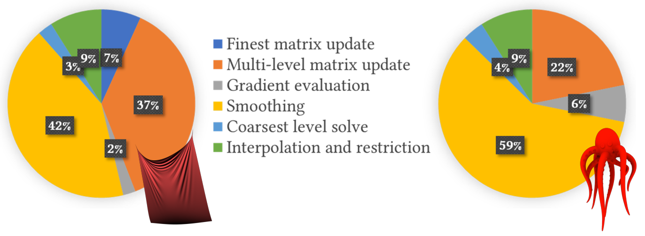

Fig. 5. Cost breakdown of method to simulate one frame of a piece of cloth (left) and an octopus (right).

Table 2 shows the cost breakdown of our method to solve for one frame of our examples. As we can see, when computing one Newton iteration, the major cost of our method is at the multi-level matrix update step where we compute all the $\mathbf{U}^\top\mathbf{A}\mathbf{U}$ matrices as described in Section 4.3 and the smoothing step where we typically run fewer than 10 Jacobi or Gauss-Seidel iterations for non-coarsest grids. In one Projective Dynamics iteration, the cost of updating the finest level matrix is close to zero because of the pre-computation of the constant system matrix. However, the cost of the multi-level matrix update is still there to account for time-varying attachment and collision constraints. The cost of gradient evaluation, coarsest level dense linear system solve, interpolation and restriction is usually small compared with the smoothing step. To illustrate this better, we pick two examples solved by Newton's method and by Projective Dynamics respectively, and visualize their cost breakdown in a pie chart as shown in Figure 5.

5.2 Validations

We design a two-grid v-cycle experiment to test the effectiveness of our multigrid method. The goal is simple: we know that stationary iterative methods take care of the high-frequency error well, and we want to see if our multigrid scheme does its duty of handling the low-frequency error. We pick an arbitrary frame from the simulation of an armadillo and setup our linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ to decide the descent direction where $\mathbf{A}$ is adopted from Projective Dynamics [Liu et al. 2017] and $\mathbf{b} = -\nabla g$. As described in Alg. 2, we first pre-smooth on the linear system using 3 iterations of symmetric Gauss-Seidel to reach an initial value $\mathbf{x}_1$, and then execute line 3 to line 7 in Alg. 2 as the coarse grid linear solve. Let us note the error after our pre-smoother as $\boldsymbol{\epsilon} = \mathbf{A}^{-1}\mathbf{b} - \mathbf{x}_1 = \mathbf{A}^{-1}\mathbf{r}_1$.

Table 1. Statistics for all our testing scenarios. Grid and solver set-up reports the number of vertices and the corresponding linear solver in each level respectively. In the solver set-up column, "GS" and "J" stands for symmetric Gauss-Seidel and Jacobi iterations with the corresponding number of iterations inside parentheses; "direct" is for a dense direct solver factorized by Cholesky decomposition which can only be used in the coarsest level. Our method solves for either a static system matrix computed from Projective Dynamics or a dynamics Hessian matrix using Newton's method. The "#iters" stands for the iteration count of the corresponding nonlinear optimization method.

example	#verts.	#elems.	grid set-up	solver set-up	system matrix	#iters	fps	pre-computation
armadillo	1241	3581	10/all	direct/GS(3)	static	1	303.0	1.9 sec
	6654	20751	25/all	direct/GS(3)	static	1	232.6	2.0 sec
	14779	54855	50/all	direct/GS(3)	static	1	181.8	3.0 sec
	50736	163779	25/500/all	direct/GS(3)/GS(3)	static	1	86.2	6.3 sec
	95587	306957	50/1000/all	direct/GS(3)/GS(3)	static	1	55.2	11.6 sec
	126966	405899	25/300/2000/all	direct/GS(3)/GS(3)/GS(3)	static	1	41.0	16.3 sec
cloth	40401	240000	25/100/1000/all	direct/J(10)/J(5)/J(5)	static	4	74.8	8.3 sec
	40401	240000	25/100/1000/all	direct/J(10)/J(5)/J(5)	dynamic	4	35.7	8.3 sec
	63001	375000	100/1000/all	direct/J(10)/J(5)	dynamic	3	31.5	13.4 sec
dragon	200094	676675	50/1000/all	direct/GS(2)/GS(2)	static	1	39.4	5.5 sec
octopus	39223	112145	100/1000/all	direct/GS(2)/GS(2)	static	3	43.2	22.8 sec

Table 2. Runtime cost breakdown of our method to generate one frame of the simulations in all our test cases. The order of examples are identical with Table 1.

example	#verts.	#elems.	finest matrix update	multi-level matrices update	gradient evaluation	smoothing	coarsest level solve	interpolation & restriction
armadillo	1241	3581	0 ms	0.29 ms	0.18 ms	2.32 ms	0.24 ms	0.26 ms
	6654	20751	0 ms	0.55 ms	0.25 ms	2.91 ms	0.28 ms	0.31 ms
	14779	54855	0 ms	1.27 ms	0.28 ms	3.30 ms	0.30 ms	0.36 ms
	50736	163779	0 ms	0.58 ms	1.05 ms	8.77 ms	0.26 ms	0.94 ms
	95587	306957	0 ms	1.36 ms	2.43 ms	12.73 ms	0.28 ms	1.30 ms
	126966	405899	0 ms	0.74 ms	3.21 ms	18.30 ms	0.25 ms	1.90 ms
cloth	40401	240000	0 ms	0 ms	0.58 ms	9.88 ms	0.67 ms	2.23 ms
	40401	240000	1.56 ms	8.35 ms	0.43 ms	8.70 ms	0.52 ms	1.92 ms
	63001	375000	2.13 ms	15.22 ms	0.73 ms	11.39 ms	0.77 ms	1.51 ms
dragon	200094	676675	0 ms	3.07 ms	1.83 ms	16.53 ms	0.50 ms	1.22 ms
octopus	39223	112145	0 ms	1.55 ms	0.05 ms	15.21 ms	0.28 ms	2.29 ms

Assuming the coarse level linear system $\mathbf{U}^\mathsf{T}\mathbf{A}\mathbf{U}\mathbf{e}_2 = \mathbf{r}_2$ is completely solved, we can find the new error after the coarse grid solve which is $\tilde{\boldsymbol{\epsilon}} = \mathbf{A}^{-1}\mathbf{b} - (\mathbf{x}_1 + \mathbf{U}\mathbf{e}_2) = \boldsymbol{\epsilon} - \mathbf{S}\boldsymbol{\epsilon}$, where $\mathbf{S}$ is defined as $\mathbf{U}\left(\mathbf{U}^\mathsf{T}\mathbf{A}\mathbf{U}\right)^{-1}\mathbf{U}^\mathsf{T}\mathbf{A}$ with a nice property: $\mathbf{S} = \mathbf{S}^2$. This property indicates the eigenvalues of $\mathbf{S}$ are either 0 or 1. Therefore $\mathbf{S}$ can be seen as an identity matrix on the subspace decided by $\mathbf{U}$ (or a zero-matrix on the complementary subspace). The intuitive explanation is that once the coarse level grid handles its coarse linear solve perfectly, a Galerkin multigrid scheme simply removes all errors in the subspace decided by $\mathbf{U}$ and ignores the rest of the errors. Of course we are not expecting $\tilde{\boldsymbol{\epsilon}} = \boldsymbol{\epsilon} - \mathbf{S}\boldsymbol{\epsilon}$ to be zero (which would be perfect, but not possible), we can at least check how much the error is reduced by defining a reduction factor $\rho = ||\tilde{\boldsymbol{\epsilon}}||/||\boldsymbol{\epsilon}||$, the lower ρ the better error reduction. We did four experiments with different set-ups to check the corresponding reduction factors ρ, as shown in Table 3, where k is the number of vertices we used for the coarse grid; *DoF* stands for the degrees of freedom for each coarse level vertex; and weights are set either discretely to piece-wise constant or smoothly using bounded biharmonic weights [Jacobson et al. 2011].

Table 3. Coarse grid error reduction (ρ) for different multigrid set-ups.

	test	k	DOF	weights	ρ
	#1	100	12	piecewise const.	0.058
	#2	100	12	smooth	0.019
	#3	100	3	piecewise const.	0.494
	#4	400	3	piecewise const.	0.407

We used an armadillo example with 14779 vertices as shown on the left of Table 3 to generate the table. Test #1 is our method which uses 100 coarse level vertices with 12 degrees of freedom for each vertex and piecewise constant weights to construct the interpolation $\mathbf{U} \in \mathbb{R}^{44337\times1200}$ matrix. We first compare our method with test #2 which uses exactly the same setting but smooth weights. As expected, the multigrid with smooth weights efficiently reduces the error to a factor of $\rho = 0.019$, as oppose to our multigrid structure with $\rho = 0.058$. However, compared to our method, this better error reduction comes at a cost of more than 5 times of memory consumption to store the $\mathbf{U}^\mathsf{T}\mathbf{A}\mathbf{U}$ matrix plus more than 10 times slower because of chunkier coarse system matrix update and denser matrix-vector multiplications. The extra memory overhead of using smooth weights would be even amplified when using multiple levels of grids.

We also tested to see whether using the skinning space coordinates for the coarse grid makes a difference. Test #3 uses our settings but gives each coarse grid vertex 3 degrees of freedom. As a result, it only cuts the error roughly by half. To make a fair comparison, we increased the number of vertices on the coarse grid to 400 when using position space coordinates, as shown in test #4. This would make the total coarse level degrees of freedom the same with our method. Increasing number of vertices helps to reduce the error more, but not as efficiently as increasing the per-vertex degrees of freedom like our method.

5.3 Scalability

Figure 6 shows a swinging armadillo example simulated using our method. The armadillo is pinned on its ears and falls under gravity. We tested multiple simulations using the same physics parameters, e.g. mass density and Lamé coefficients, on different resolutions of the same armadillo, in order to see the scalability of our method. Our method shows consistent motions of the armadillo, despite the tessellation difference for different meshes. That indicates that our method solves the linear systems well, regardless of the increase of resolution. As reported in Table 1, all the armadillo examples are simulated in real-time.

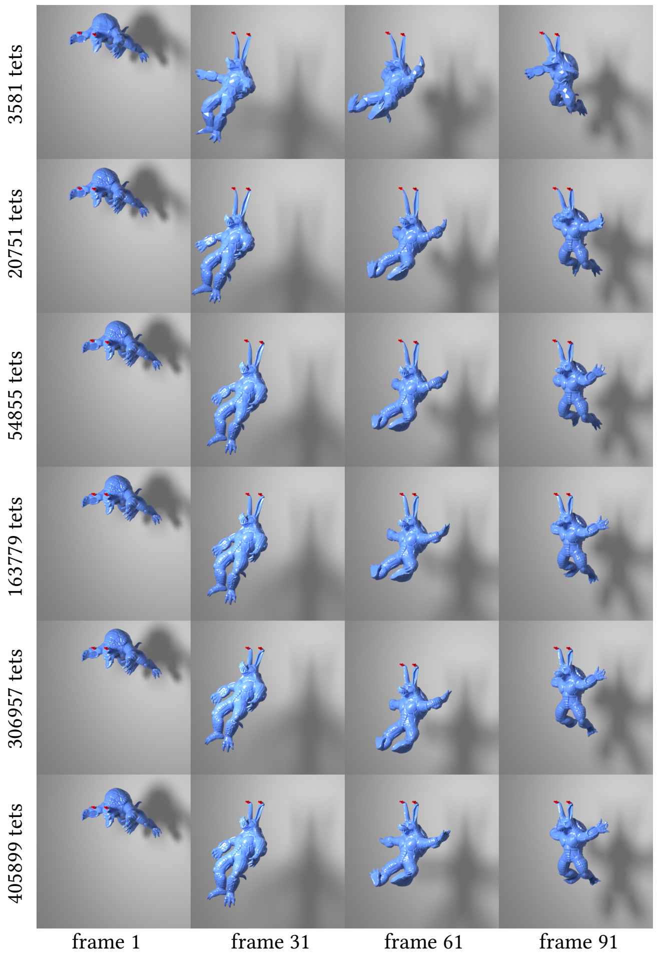

Fig. 6. The same armadillo example with different resolutions ranging from 3581 tetrahedra to 405899 tetrahedra.

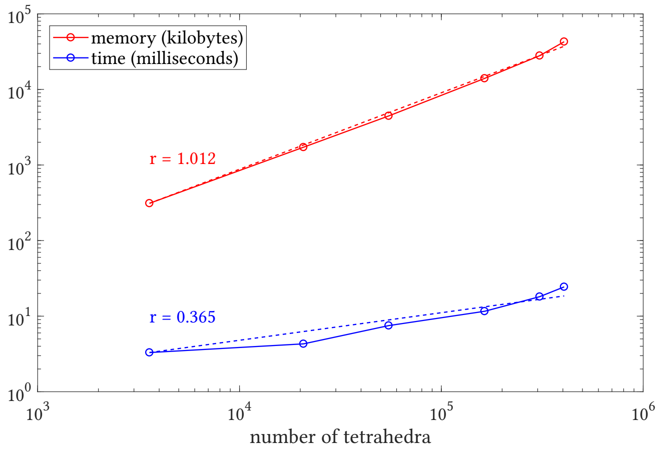

Fig. 7. Scalability of our method. We executed the armadillo example with different resolutions ranging from 10^3 to 10^5 tetrahedra, measuring the cost of memory (red graph) and time (blue graph) for each simulation. The dashed line with slope r shows the slope after running a linear regression on this log-log scaled plot.

To numerically validate the scalability of our method, we also show the cost of memory and time of the armadillo examples in Figure 7. The memory cost graph (red graph) plots the extra memory overhead to store the interpolation matrices $\mathbf{U}$s, the linear system matrices $\mathbf{U}^T\mathbf{A}\mathbf{U}$s on different level grids and the coarsest level factors from a dense Cholesky factorization. The time graph (blue graph) shows the time to simulate an entire frame. We compute the slopes of best-fit lines on this log-log scaled figure to see our scalability. Our memory cost grows linearly (with a slope $r = 1.012$) with the number of elements because of the linear growth of $\mathbf{U}$ and $\mathbf{U}^T\mathbf{A}\mathbf{U}$ matrices. Performance-wise, our method scales sub-linearly (with only a slope $r = 0.365$) with respect to the mesh resolution, fully taking the advantages of the multigrid hierarchy. The sub-linear performance scale is also a result of our GPU implementation where higher resolution simulations take more advantages from the parallelization.

5.4 Flexibility

As a general linear solver, our multigrid method is flexible that can be equipped with different descent methods. Figure 8 shows an example of a hanging cloth simulation. Our method can be used to replace the direct solver of Projective Dynamics [Bouaziz et al. 2014; Liu et al. 2013]. The first row shows a GPU implementation of Projective Dynamics using a pre-factorized Cholesky solver. Due to the parallelization unfriendly nature of forward/backward substitution,

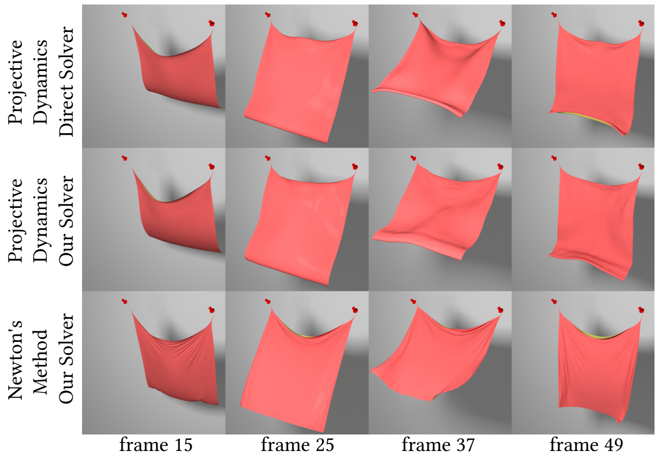
Projective Dynamics Direct Solver
Projective Dynamics Our Solver
Newton's Method Our Solver
frame 15
frame 25
frame 37
frame 49

Fig. 8. Hanging cloth example with different numerical solvers. The first row is simulated using Projective Dynamics solved using pre-factorized direct solver; the second row is Projective Dynamics solved using our multigrid solver; the third row is Newton's method solved using our solver.

and bad memory consumption of the pre-computed factors, Projective Dynamics runs at 9.2 frames per second. Our method can be used to accelerate Projective Dynamics, achieving 74.8 frames per second which is an 8-times speed-up. However, under the Projective Dynamics setting, our method also inherits its artifacts such as bad wrinkles due to slow convergence for high-frequency components, as we can see from the second row of Figure 8. Our method does not depend on a specific choice of the system matrix, therefore, can be applied with Newton's method as well. The third row of Figure 8 shows the result of the cloth simulated using Newton's method solved by our multigrid technique. The frame rate dropped to 35.7 because of the run-time construction of the Hessian matrix and the update of multi-level system matrices. Note that our multigrid for Newton's method still runs at real-time, while produces much better wrinkle effects compared to Projective Dynamics.

5.5 Varying stiffness parameters

Increasing the stiffness of a system always make that system harder to solve because of two reasons – it increases the condition number of the linear systems and increase the nonlinearity of the entire optimization problem. We tested our linear solver with different stiffness parameters ranging from 5×10^2 to 2.56×10^5 as shown in Figure 9. We measured our results by the normalized residual which is simply $||\mathbf{Ax} - \mathbf{b}|| / ||\mathbf{Ax}_0 - \mathbf{b}||$ when solving $\mathbf{Ax} = \mathbf{b}$ with initial guess $\mathbf{x}_0$. All the experiments show similar residual reduction factors, indicating that the accuracy of our method to solve linear systems is hardly affected by varying stiffness parameters. The increase of nonlinearity of the entire optimization problem will require most nonlinear optimization methods more iterations to converge, a similar conclusion was reported by Liu and colleagues[2013]. With our acceleration scheme for linear solvers, we are able to run more nonlinear iterations to better support stiff materials within a certain time budget.

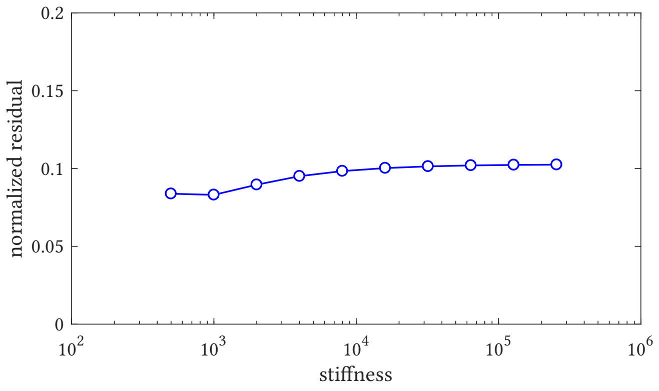

Fig. 9. The normalized residual of our multigrid linear solver when solving systems with varying stiffness parameters.

5.6 Comparisons

We compare our method with other parallel iterative solvers for deformable body simulation [Fratarcangeli et al. 2018] in both 2-dimensional and 3-dimensional scenarios as shown in Figure 10 and in Figure 11 respectively. We tested three linear solvers – the Vivace solver [Fratarcangeli et al. 2016], the Chebyshev semi-iterative solver [Wang 2015] and our multigrid solver with the same linear system from Projective Dynamics [Bouaziz et al. 2014] to make a fair comparison. All three solvers are implemented in GPU. From the same starting frame, we compare the convergence behavior of all three methods with the following relative error:

$$\frac{g(\mathbf{x}^{(k)}) - g(\mathbf{x}^*)}{g(\mathbf{x}^{(0)}) - g(\mathbf{x}^*)} \tag{9}$$

where $\mathbf{x}^*$ is the ground truth solution computed using Newton's method (iterated until convergence), $\mathbf{x}^{(0)}$ is the initial guess and $\mathbf{x}^{(k)}$ is the result after k-th iterate of Projective Dynamics.

The Vivace solver was tested twice using different numbers of symmetric Gauss-Seidel iterations (10 and 50). We always started the Chebyshev semi-iterative solver with 1 Jacobi iteration, and

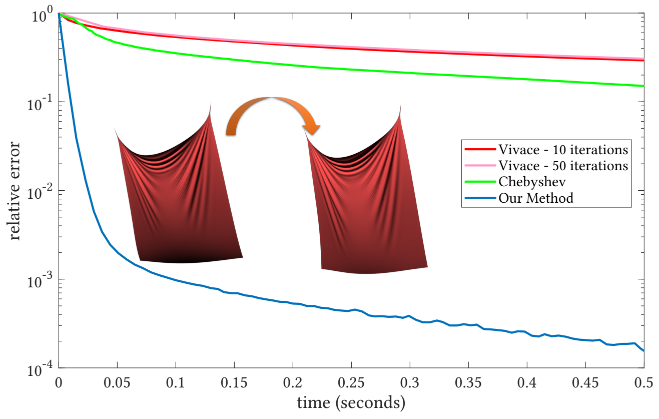

Fig. 10. Comparison on a cloth simulation with 40401 vertices and 240000 springs.

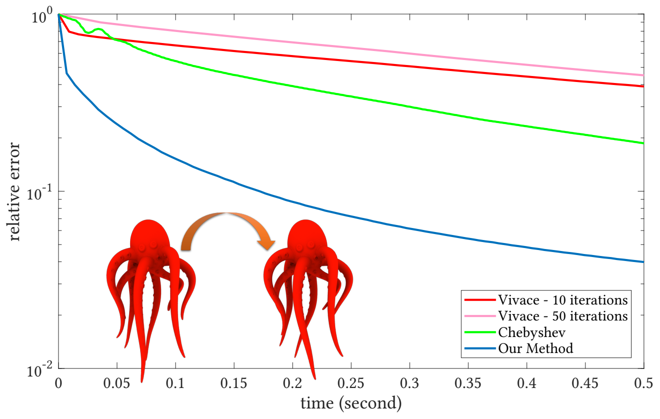

Fig. 11. Comparison on an octopus simulation with 39223 vertices and 112145 tetrahedra.

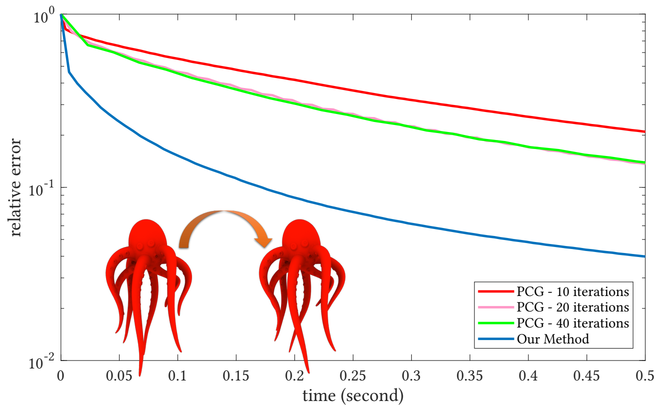

Fig. 12. Comparison with preconditioned conjugate gradient methods on an octopus simulation with 39223 vertices and 112145 tetrahedra.

we only enabled the Chebyshev acceleration after 10 iterations as suggested by Wang[2015]. We fine-tuned the ρ parameter for the Chebyshev method to make it fit best to specific examples.

In both of the cloth and octopus cases, our method outperforms the other iterative solver competitors. Note that the first x-axis tick (0.05 seconds) in Figure 10 and Figure 11 matters the most in real-time applications. Failing to sufficiently reduce the error inside this first tick might produce visually unacceptable artifacts, e.g. over-stretch of the simulated object. Please refer to our accompanying video to see the difference between three iterative solvers better.

We also compare our method with a Jacobi-preconditioned conjugate gradient (PCG) method which is a straight-forward solution to implement in parallel. The results can be seen in Figure 12. PCG runs reasonably well, despite the reduction and synchronization overhead due to the frequent inner products in the algorithm. We tested our method against different PCG settings with 10, 20 and 40 iterations per linear solve. Our method work best among all our test scenarios.

6 LIMITATIONS AND FUTURE WORK

Compared to other stationary iterative methods, our multigrid acceleration, of course, does not come as a free lunch. The speed-up of our method greatly relies on our precomputation where the construction of **U** matrices takes place. If the topology of the mesh is changed, for instance when an object is torn apart, we need to resample the multi-level grid vertices and compute the corresponding interpolation matrices **U**.

Our method currently works only for homogeneous materials because we uniformly sampled the multi-resolution vertices before constructing our **U** matrices. It would be interesting to investigate a representative sampling method for heterogeneous materials.

Our multigrid scheme can handle only soft constraints for now. This is an inherited problem from the Galerkin projection. An intuitive explanation is that gradient (or force) can be efficiently passing back and forth between different grid levels by projection, but the that does not apply for hard constraints. We would like to investigate the possibility to support hard constraints for elastic body simulations [Goldenthal et al. 2007; Müller et al. 2007], or a more general complaint-constraint-based scheme that treats both soft and hard constraints in a unified framework [Macklin et al. 2016; Tournier et al. 2015].

Another perfect direction to explore is collisions. In spite of the possibility to support hard collision constraints like we just mentioned, accelerating the collision detection, especially for self collisions is an interesting research problem. Thanks to the piecewise constant weighting, our multigrid method naturally provides a forest-like structure which could be helpful for fast multi-resolution collision detections.

In all our experiments, we only use our multigrid as a standalone linear solver. Our multigrid solver does not break the symmetry of the system matrix because of our choice of symmetric smoothers such as Jacobi or symmetric Gauss-Seidel. Therefore, it would be an interesting extension to use our multigrid as a preconditioner for a PCG solver when dealing with more challenging cases.

Multigrid methods are a lot of fun, we hope our multigrid scheme would inspire more work on real-time large-scaled simulations.

ACKNOWLEDGMENTS

We thank Alec Jacobson, Yang Liu and Eftychios Sifakis for many insightful discussions, Huamin Wang for providing his GPU simulation framework, and the anonymous reviewers for their valuable feedbacks.

REFERENCES

David Arthur and Sergei Vassilvitskii. 2007. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1027–1035.

David Baraff and Andrew Witkin. 1998. Large steps in cloth simulation. In *Proc. of ACM SIGGRAPH*. 43–54.

Jernej Barbič and Doug L James. 2005. Real-time subspace integration for St. Venant-Kirchhoff deformable models. In *ACM Trans. Graph.*, Vol. 24. ACM, 982–990.

Jan Bender, Matthias Müller, and Miles Macklin. 2015. Position-Based Simulation Methods in Computer Graphics.. In *Eurographics (Tutorials)*.

Jan Bender, Matthias Müller, Miguel A Otaduy, Matthias Teschner, and Miles Macklin. 2014. A survey on position-based simulation methods in computer graphics. In *Comput. Graph. Forum*, Vol. 33. Wiley Online Library, 228–251.

Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2014. Projective dynamics: fusing constraint projections for fast simulation. *ACM Trans.*

Graph. 33, 4 (2014), 154.
Christopher Brandt, Elmar Eisemann, and Klaus Hildebrandt. 2018. Hyper-reduced projective dynamics. *ACM Trans. Graph.* 37, 4 (2018), 80.
Min Gyu Choi and Hyeong-Seok Ko. 2005. Modal warping: Real-time simulation of large rotational deformation and manipulation. *IEEE Transactions on Visualization and Computer Graphics* 11, 1 (2005), 91–101.
Christian Dick, Joachim Georgii, and Rüdiger Westermann. 2011. A real-time multigrid finite hexahedra method for elasticity simulation using CUDA. *Simulation Modelling Practice and Theory* 19, 2 (2011), 801–816.
Marco Fratarcangeli, Valentina Tibaldo, and Fabio Pellacini. 2016. Vivace: A practical gauss-seidel method for stable soft body dynamics. *ACM Trans. Graph.* 35, 6 (2016), 214.
Marco Fratarcangeli, Huamin Wang, and Yin Yang. 2018. Parallel iterative solvers for real-time elastic deformations. In *SIGGRAPH Asia 2018 Courses*. ACM, 14.
Lawson Fulton, Vismay Modi, David Duvenaud, David I. W. Levin, and Alec Jacobson. 2019. Latent-space Dynamics for Reduced Deformable Simulation. *Comput. Graph. Forum* (2019).
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 (2015), 1103–1115.
Joachim Georgii and Rüdiger Westermann. 2006. A multigrid framework for real-time simulation of deformable bodies. *Computers & Graphics* 30, 3 (2006), 408–415.
Rony Goldenthal, David Harmon, Raanan Fattal, Michel Bercovier, and Eitan Grinspun. 2007. Efficient simulation of inextensible cloth. *ACM Trans. Graph.* 26, 3 (2007), 49.
Alec Jacobson. 2015. How does Galerkin multigrid scale for irregular grids? http://www.alecjacobson.com/weblog/?p=4383
Alec Jacobson, Ilya Baran, Ladislav Kavan, Jovan Popović, and Olga Sorkine. 2012. Fast automatic skinning transformations. *ACM Trans. Graph.* 31, 4 (2012), 77.
Alec Jacobson, Ilya Baran, Jovan Popovic, and Olga Sorkine. 2011. Bounded biharmonic weights for real-time deformation. *ACM Trans. Graph.* 30, 4 (2011), 78–1.
Inyong Jeon, Kwang-Jin Choi, Tae-Yong Kim, Bong-Ouk Choi, and Hyeong-Seok Ko. 2013. Constrainable multigrid for cloth. In *Comput. Graph. Forum*, Vol. 32. Wiley Online Library, 31–39.
Liliya Kharevych, Weiwei Yang, Yiying Tong, Eva Kanso, Jerrold E Marsden, Peter Schröder, and Matthieu Desbrun. 2006. Geometric, variational integrators for computer animation. *Proc. EG/ACM Symp. Computer Animation*, 43–51.
Minchen Li, Ming Gao, Timothy Langlois, Chenfanfu Jiang, and Danny M Kaufman. 2019. Decomposed optimization time integrator for large-step elastodynamics. *ACM Trans. Graph.* 38, 4 (2019), 70.
Haixiang Liu, Yuanming Hu, Bo Zhu, Wojciech Matusik, and Eftychios Sifakis. 2018. Narrow-band topology optimization on a sparsely populated grid. In *Proc. of ACM SIGGRAPH Asia*. ACM, 251.
Tiantian Liu, Adam W Bargteil, James F O'Brien, and Ladislav Kavan. 2013. Fast simulation of mass-spring systems. *ACM Trans. Graph.* 32, 6 (2013), 214.
Tiantian Liu, Sofien Bouaziz, and Ladislav Kavan. 2017. Quasi-newton methods for real-time simulation of hyperelastic materials. *ACM Trans. Graph.* 36, 4 (2017), 116a.
Miles Macklin and Matthias Müller. 2013. Position based fluids. *ACM Trans. Graph.* 32, 4 (2013), 104.
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, 49–54.
Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. 2014. Unified particle physics for real-time applications. *ACM Trans. Graph.* 33 (2014), 153:1–153:12.
Nadia Magnenat-Thalmann, Richard Laperrire, and Daniel Thalmann. 1988. Joint-dependent local deformations for hand animation and object grasping. In *In Proceedings on Graphics interface'88*. Citeseer.
Sebastian Martin, Bernhard Thomaszewski, Eitan Grinspun, and Markus Gross. 2011. Example-based elastic materials. In *ACM Trans. Graph.*, Vol. 30. ACM, 72.
Aleka McAdams, Yongning Zhu, Andrew Selle, Mark Empey, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. 2011. Efficient elasticity for character skinning with contact and collisions. *ACM Trans. Graph.* 30, 4 (2011), 37.
Matthias Müller. 2008. Hierarchical Position Based Dynamics. In *Workshop in Virtual Reality Interactions and Physical Simulation "VRIPHYS" (2008)*. The Eurographics Association.
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. In *ACM Trans. Graph.*, Vol. 24. 471–478.
Matthew Overby, George E Brown, Jie Li, and Rahul Narain. 2017. ADMM ⊇ Projective Dynamics: Fast Simulation of Hyperelastic Models with Dynamic Constraints. *IEEE TVCG* 23, 10 (2017), 2222–2234.
A.R. Rivers and D.L. James. 2007. FastLSM: fast lattice shape matching for robust real-time deformation. *ACM Trans. Graph.* 26 (2007), 82:1–82:6.
Eftychios Sifakis and Jernej Barbic. 2012. FEM simulation of 3D deformable solids: a practitioner's guide to theory, discretization and model reduction. In *ACM SIGGRAPH 2012 courses*. ACM, 20.
Breannan Smith, Fernando De Goes, and Theodore Kim. 2019. Analytic Eigensystems for Isotropic Distortion Energies. *ACM Trans. Graph.* 38, 1 (2019), 3.
Jos Stam. 2009. Nucleus: towards a Unified Dynamics Solver for Computer Graphics. In *IEEE Int. Conf. on CAD and Comput. Graph.* 1–11.
Ari Stern and Mathieu Desbrun. 2006. Discrete geometric mechanics for variational time integrators. In *ACM SIGGRAPH Courses*. ACM, 75–80.
Gilbert Strang and Kaija Aarikka. 1986. *Introduction to applied mathematics*. Vol. 16. Wellesley-Cambridge Press Wellesley, MA.
Rasmus Tamstorf, Toby Jones, and Stephen F McCormick. 2015. Smoothed aggregation multigrid for cloth simulation. *ACM Trans. Graph.* 34, 6 (2015), 245.
Joseph Teran, Eftychios Sifakis, Geoffrey Irving, and Ronald Fedkiw. 2005. Robust quasistatic finite elements and flesh simulation. In *Proc. EG/ACM Symp. Computer Animation*. ACM, 181–190.
Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. 1987. Elastically deformable models. In *Computer Graphics (Proceedings of SIGGRAPH)*, Vol. 21. 205–214.
Maxime Tournier, Matthieu Nesme, Benjamin Gilles, and Francois Faure. 2015. Stable constrained dynamics. *ACM Trans. Graph.* 34, 4 (2015), 132.
Christoph Von Tycowicz, Christian Schulz, Hans-Peter Seidel, and Klaus Hildebrandt. 2013. An efficient construction of reduced deformable objects. *ACM Trans. Graph.* 32, 6 (2013), 213.
Huamin Wang. 2015. A chebyshev semi-iterative approach for accelerating projective and position-based dynamics. *ACM Trans. Graph.* 34 (2015), 246:1–246:9.
Yu Wang, Alec Jacobson, Jernej Barbič, and Ladislav Kavan. 2015. Linear subspace design for real-time shape deformation. *ACM Trans. Graph.* 34, 4 (2015), 57.
Zhendong Wang, Longhua Wu, Marco Fratarcangeli, Min Tang, and Huamin Wang. 2018. Parallel Multigrid for Nonlinear Cloth Simulation. In *Comput. Graph. Forum*, Vol. 37. Wiley Online Library, 131–141.
Jun Wu, Christian Dick, and Rudiger Westermann. 2016. A System for High-Resolution Topology Optimization. *IEEE TVCG* 22, 3 (2016), 1195–1208.
Yufeng Zhu, Robert Bridson, and Danny M Kaufman. 2018. Blended cured quasi-newton for distortion optimization. *ACM Trans. Graph.* 37, 4 (2018), 40.
Yongning Zhu, Eftychios Sifakis, Joseph Teran, and Achi Brandt. 2010. An efficient multigrid method for the simulation of high-resolution elastic solids. *ACM Trans. Graph.* 29, 2 (2010), 16.

APPENDIX

As reported in Table 4, we show the simulation parameters we used to produce all our results in this paper and in the accompanying video. The mass density represents area density for cloth and volume density for other volumetric meshes. The material stiffness column reports the stretch stiffness and bending stiffness for mass-spring systems and Lamé's second coefficient μ and first coefficient λ for finite element models.

Table 4. Simulation parameters.

example	mass density	material model	material stiffness	attachment stiffness	collision stiffness
armadillo	1.0	corot	500/0	1	1
cloth	1.0	mass-spring	100/20	100	1
dragon	1.0	corot	250/0	1	-
octopus	1.0	corot	1000/0	1	-