Efficient Elasticity for Character Flesh Simulation on Octrees

Andy Milne [1] Mark A. McLaughlin [1] Haixiang Liu [1,2] Alexey Stomakhin [1] Rasmus Tamstorf [1]

[1] Walt Disney Animation Studios [2] University of Wisconsin, Madison

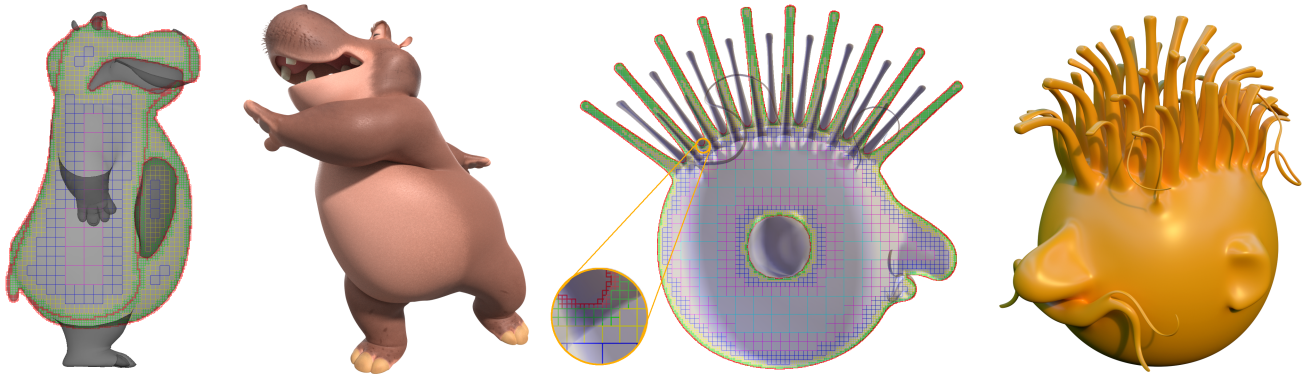


Figure 1: *(a) Slice through the center of hippo octree. (b) Simulated hippo character. (c) Slice through center of Spike octree. (d) Simulated Spike character.*

Abstract

We present an efficient method for the physical simulation of soft tissue deformation for animated characters using octrees. Previous work on efficient elasticity simulation relies on the regular structure of a uniform grid discretization. However, representing the complicated boundaries of animated characters requires a high resolution grid, which consumes large amounts of memory and is computationally expensive. By refining an octree around the surface of the character, our memory usage and simulation times grow only quadratically with the inverse of the cell size, compared to cubic growth for a uniform grid. We realize memory savings of 18x and a speedup of 1.7x over a uniform grid with the same boundary resolution on one production example. Moreover, we show that we are able to simulate complex geometry that is infeasible with a uniform grid due to memory usage that would exceed our method by 50x or more. To achieve this we derive a matrix-free geometric multigrid method for the solution of linear systems resulting from an octree discretization of the equations of corotational linear elasticity. We also demonstrate the high parallel scalability of our optimized shared-memory CPU implementation in a fully-featured production-level system.

Keywords: simulation, physically based animation, skinning, corotated elasticity, finite element method, octree, multigrid

Concepts: •**Computing methodologies** → **Physical simulation;**

1 Introduction

Physical simulation is now standard in computer animated feature films. However, simulation on animated characters is often confined to hair and clothing, and is less frequently used for the body of the character itself. Simulation of the deformation and dynamics of soft tissues such as skin, muscles and fat is more common in visual effects, where realism is paramount [Clutterbuck and Jacobs 2010; Comer et al. 2015; McLaughlin et al. 2011; Schutz et al. 2016]. In animated feature films, flesh simulation has been used in situations where it is impossible to achieve a satisfactory aesthetic result by other means, typically on a small number of characters or shots per film [Irving et al. 2008; Kautzman et al. 2012; Kautzman et al. 2016]. Although flesh simulation has recently been used on a larger scale in animated feature films [Milne et al. 2016b], it is still standard industry practice to use non-simulation-based techniques for skin deformation, such as dual-quaternion skinning [Kavan et al. 2008] and pose-space deformation [Lewis et al. 2000]. However, there is a growing demand for the rich organic motion that simulation provides.

The finite element method (FEM) is the method of choice for simulating elastic materials such as flesh. Until recently, running FEM flesh simulations on every character in every shot of an animated feature film has been considered too expensive to be feasible. A method for efficient elasticity simulation for animated characters was presented by [McAdams et al. 2011]. Their method embeds the character in a hexahedral lattice. The uniform grid structure is computationally efficient, and allows the use of a matrix-free geometric multigrid method with a direct coarse grid discretization for solving the resulting equations, further improving performance.

However, a uniform grid is unsatisfactory in two aspects. In order to realize the benefits of the simple and efficient element and node indexing permitted by a uniform grid, it must be stored as a 3D array. Embedding the character in such a box-shaped domain wastes memory, as large portions of the grid are empty. In addition, the grid cells must be sufficiently small to resolve the boundary of the character accurately. Using such small cells over the entire domain may be prohibitively expensive in terms of memory usage and computation time.

To address both of these issues, we propose to extend the method of McAdams et al. to octrees. We embed the character in an incomplete octree, which is refined around the boundary. For efficiency, we use a linear, pointerless octree encoding. Our geometric multigrid method on octrees is completely matrix-free and uses a direct coarse grid discretization. This requires less than one quarter of the memory as multigrid methods based on Galerkin coarsening, which must store a sparse matrix on each multigrid level [Milne 2015]. Since memory bandwidth is a bottleneck in most sparse matrix computations [Georgii and Westermann 2010], this can also significantly impact speed. Our method has the potential to enable the

widespread use of flesh simulation on characters in animated feature films.

The rest of this paper is organized as follows. After discussing related work, we describe in section 3 our approach to the fundamental challenge of applying the FEM to octrees, which is the existence of "hanging nodes". In section 4 we review corotational linear elasticity along with the stabilized one-point quadrature we use. We describe our linear octree data structure in section 5 and octree construction in section 6. In section 7, we specify the components of our matrix-free geometric multigrid method, including octree coarsening, restriction and prolongation, and the formulation of the coarse grid operator. We detail how we compute the diagonal part of the stiffness matrix, which is needed for the multigrid smoother but not available in a matrix-free method. Section 8 discusses the incorporation of constraints and collisions. We give some details of our parallel implementation in section 9. Finally, we present results in section 10 and conclude.

2 Related Work

More details on some aspects of this work can be found in non-peer-reviewed prepublications by the authors [Milne 2015; Milne et al. 2016a].

[McAdams et al. 2011] presented a discretization of corotational elasticity on a uniform hexahedral lattice with a stabilized one-point quadrature. This is more efficient than the standard 8-point Gaussian quadrature, while avoiding the unstable hourglass modes of a naive one-point quadrature. We show that this technique extends naturally to octrees. [Patterson et al. 2012] extended the stabilization technique of [McAdams et al. 2011] to constitutive models other than corotational linear elasticity, but we do not pursue that further in this paper.

[Braess 1986; McAdams et al. 2010; Sampath et al. 2008] used multigrid as a preconditioner to the conjugate gradient method (CG) instead of as stand-alone solver. This is advantageous because it allows the use of a simple multigrid method, while a more expensive and sophisticated multigrid method may be required for good convergence as a stand-alone solver for some problems. We intentionally use a simplified matrix-free geometric method because it is highly efficient. Accordingly, we have found that using multigrid as a preconditioner gives faster convergence in our application. Slightly improved convergence rates were also observed by [Dick et al. 2011], although they found that the benefit was not worth the increased cost. In contrast, [Sundar et al. 2014] found that using multigrid as a preconditioner converged significantly faster than multigrid alone.

[Seiler et al. 2010; Dick et al. 2011] solved corotational linear elasticity on octrees using a warped stiffness formulation. [Seiler et al. 2010] assembled a global stiffness matrix, which they solved with preconditioned CG. [Dick et al. 2011] used a geometric multigrid method with Galerkin coarsening. Their method is matrix-free in the sense that they did not assemble a global stiffness matrix, but the use of Galerkin coarsening means that the method can only be fully matrix-free at the finest multigrid level. On the coarser levels, they stored per-element stiffness matrices.

Many fast techniques for soft body simulation have been proposed for interactive applications, often targeting GPU implementations. Projective dynamics [Bouaziz et al. 2014] is a popular method that has been applied to character deformation [Lin et al. 2016]. [Fratarcangeli et al. 2016] presented a randomized Gauss-Seidel solver suitable for implementation of projective dynamics on the GPU. [Wang and Yang 2016] presented a gradient descent method that requires no dot products.

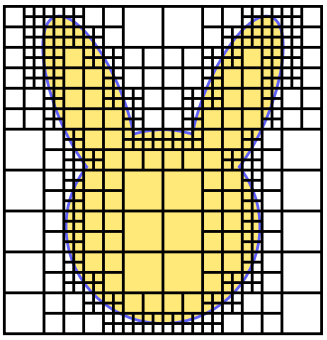


Figure 2: *A 2D slice through an octree refined around the boundary of an embedded elastic object.*

Others have proposed to speed up FEM simulations by reducing the number of degrees of freedom. [Mitchell et al. 2016] aggregated elements into macroblocks. [Torres et al. 2016] introduced a novel coarsening method for heterogeneous materials.

We use a linear octree data structure with octants sorted by Morton code as in [Sundar et al. 2008; Sampath et al. 2008; Sundar et al. 2007; Sampath and Biros 2010]. While these works used complete octrees, we embed the character in an incomplete octree which omits the empty cells. [Flaig and Arbenz 2012] used a linear data structure with a uniform cell size that omitted empty cells. [Dick et al. 2011] used pointer-based incomplete octrees. We use the efficient algorithm of [Chan 2002] to compare Morton codes.

Octrees contain "hanging nodes" or "T-vertices" in the middle of edges and faces where smaller elements are adjacent to larger elements. These must be carefully handled in order to enforce the continuity of the solution between elements of different sizes. We use the method proposed by [Wang 2000] for 2D and extended by [Sundar et al. 2007] to 3D. We use the formulation in [Dick et al. 2011], where unknowns at hanging nodes are substituted with interpolated values from the adjacent nonhanging nodes.

[Sundar et al. 2007; Sundar et al. 2008; Burstedde et al. 2011; Isaac et al. 2012] developed methods for constructing, balancing, and performing finite element computations on linear octrees in parallel. [Sampath et al. 2008; Sampath and Biros 2010; Flaig and Arbenz 2012] presented parallel matrix-free geometric multigrid methods on linear octrees. We draw heavily on all of this work. These papers solved linear elasticity problems only, while we solve corotational linear elasticity to handle large deformations.

3 Finite Element Method on Octrees

In this section, we describe our approach to applying the FEM to octrees. Although we simulate 3D elasticity, figures are in 2D for clarity. We begin by introducing some notation. The domain of the elastic object is $\Omega \subset \mathbb{R}^3$. Undeformed material points are denoted $\boldsymbol{X} = (X^{(1)}, X^{(2)}, X^{(3)})^T$. The deformation function $\boldsymbol{\phi} : \Omega \times [0, \infty) \to \mathbb{R}^3$ maps each material point $\boldsymbol{X}$ at time t to its deformed location $\boldsymbol{x} = \boldsymbol{\phi}(\boldsymbol{X}, t)$.

3.1 Discretization

We discretize the domain Ω into regular hexahedra. In an octree, hexahedral elements are selectively and recursively refined into 8 children (figure 2). Compared to unstructured adaptive discretizations, octrees retain much of the simplicity and efficiency of regular grids. However, octrees contain "hanging nodes" where elements of

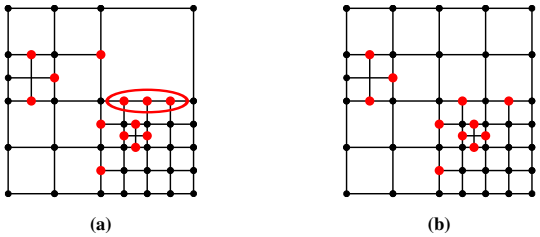


Figure 3: *2D slice through octrees with hanging nodes in red. (a) Unbalanced octree. Hanging nodes violating the 2:1 rule are circled. (b) The octree has been 2:1 balanced by splitting the largest element.*

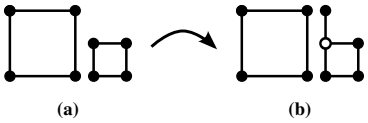


Figure 4: *Exploded view of adjacent elements of different sizes. (a) Unmodified smaller element has a hanging node. (b) Modified element with an empty circle indicating the hanging node has been removed as a degree of freedom and replaced with a node from the adjacent larger element.*

different sizes meet. Properly handling hanging nodes is the biggest challenge of using an octree discretization.

We require our octrees to be "2:1 balanced" (figure 3). This means that the ratio of the edge lengths of adjacent elements may not exceed 2:1. Limiting the rate of mesh gradation improves the mesh quality and the convergence of the FEM. It also implies that each edge or face may have at most one hanging node, which limits the number of cases that must be considered and simplifies many algorithms.

The domain of element e is Ω_e, its edge length is h and its volume is $V_e = h^3$. Each element has 8 nodes at the corners, which we call "primary nodes" to distinguish them from other nodes to be introduced later. The undeformed position of the primary node a is $\boldsymbol{X}_a$ and the deformed position is $\boldsymbol{x}_a$. Inside the element, the deformation is interpolated using trilinear shape functions. The "unmodified" shape function N_a has a value of 1 at the primary node a and a value of 0 at all other primary nodes. In order to express the shape functions in the same form on each element, we parameterize the elements by $\boldsymbol{\xi} = (\xi^{(1)}, \xi^{(2)}, \xi^{(3)})^T \in [0,1]^3$. The deformed position of a point in an element is

$$\boldsymbol{x}(\boldsymbol{\xi}) = \sum_a \boldsymbol{x}_a N_a(\boldsymbol{\xi}). \tag{1}$$

The deformation gradient $\boldsymbol{F} \triangleq \partial \boldsymbol{x} / \partial \boldsymbol{X}$ at a point in an element is

$$F_{ij}(\boldsymbol{\xi}) = \left.\frac{\partial x^{(i)}}{\partial X^{(j)}}\right|_{\boldsymbol{\xi}} = \frac{1}{h}\left.\frac{\partial x^{(i)}}{\partial \xi^{(j)}}\right|_{\boldsymbol{\xi}} = \frac{1}{h}\left.\sum_a x_a^{(i)} \frac{\partial N_a}{\partial \xi^{(j)}}\right|_{\boldsymbol{\xi}}. \tag{2}$$

3.2 Element Modification

At the location of a hanging node, the larger element has no degrees of freedom (figure 4a) and the deformation is interpolated using the element's shape functions. But the interpolated value may not match the value at the hanging node of the smaller element. To enforce continuity, we follow the approach of [Wang 2000; Sundar et al. 2007; Dick et al. 2011] and modify elements containing hanging nodes (figure 4b). The hanging node is not an independent degree of freedom. Instead, the hanging node points to a nonhanging node on the adjacent larger element (figure 5). This has the effect of modifying the shape functions on the element containing the hanging node (figure 6).

Accordingly, each element has two sets of nodes associated with it. The 8 "primary nodes" at the corners of the unmodified element may or may not be hanging. Each primary node a of element e has an associated nonhanging node $b = \mathcal{C}^e(a)$ on the modified element, which we call the "conforming node". We use the parameterization $\boldsymbol{\xi} \in [0,1]^3$ for the cubic region bounded by the primary nodes. At the conforming nodes, $\boldsymbol{\xi}$ may have values outside of the range $[0,1]^3$. The "modified" shape function $\tilde{N}_b$ has a value of 1 at the associated conforming node and a value of 0 at all other conforming nodes. Both N_a and $\tilde{N}_b$ are trilinear functions.

We prefer to work with the primary nodes and unmodified shape functions because the formulas have the same form on all elements. We can derive the relationship between the modified and unmodified shape functions by enforcing continuity at the primary nodes. Let $[\tilde{\boldsymbol{x}}_b]$ be the deformed position of the conforming nodes, and $[\boldsymbol{x}_a]$ be the interpolated values at the primary nodes. Let $\boldsymbol{\xi}_a$ be the parameter of the primary node a. We define the 8×8 matrix $\boldsymbol{W}$ by

$$W_{ab} = \tilde{N}_b(\boldsymbol{\xi}_a).$$

Multiplication by $\boldsymbol{W}$ effectively implements a gather operation, while multiplication by $\boldsymbol{W}^T$ implements the corresponding scatter operation. Using this,

$$\boldsymbol{x}_a = \boldsymbol{x}(\boldsymbol{\xi}_a) = \sum_b \tilde{\boldsymbol{x}}_b \tilde{N}_b(\boldsymbol{\xi}_a) = \sum_b W_{ab} \tilde{\boldsymbol{x}}_b.$$

This allows us to use formulas written in terms of primary nodes, such as (2) for the deformation gradient, without modification. Many of the weights are zero or one, so rather than computing $\boldsymbol{W}$ and implementing the full summation, we store a compact representation of the hanging node configuration of the element.

We can write the deformed position at a point in the element in terms of both the modified and unmodified shape functions

$$\begin{aligned}\boldsymbol{x}(\boldsymbol{\xi}) &= \sum_b \tilde{\boldsymbol{x}}_b \tilde{N}_b(\boldsymbol{\xi}) \\ &= \sum_a \boldsymbol{x}_a N_a(\boldsymbol{\xi}) = \sum_{a,b} W_{ab} \tilde{\boldsymbol{x}}_b N_a(\boldsymbol{\xi}).\end{aligned}$$

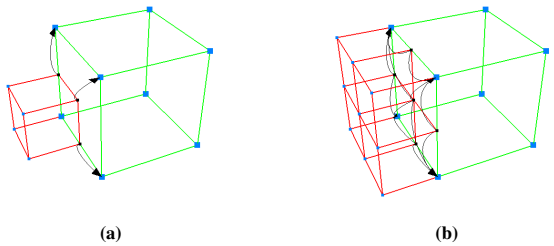


Figure 5: *A larger element surrounded by smaller elements. Nonhanging nodes are shown in blue. Hanging nodes are shown in black with arrows pointing to their nonhanging node. (a) A small element with 2 edge-hanging and 1 face-hanging nodes. (b) 4 small elements with 4 edge-hanging and 1 face-hanging nodes. Note that the same hanging node on different elements points to different nonhanging nodes.*

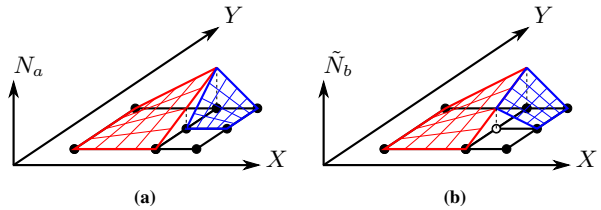


Figure 6: *(a) Unmodified shape functions have different values at the hanging node. (b) The hanging node has been removed as a degree of freedom from the smaller element and replaced with an adjacent node on the larger element. The shape function of the smaller element has been modified accordingly.*

Since we require this to hold identically for all values of $\tilde{\boldsymbol{x}}_b$,

$$\tilde{N}_b(\boldsymbol{\xi}) = \sum_a W_{ab} N_a(\boldsymbol{\xi}).$$

Thus the values of the modified shape functions can be computed from the unmodified shape functions using $\boldsymbol{W}^T$.

We compute forces at the primary nodes and distribute them to the conforming nodes. The force on the primary node a due to an energy E is

$$f_a^{(i)} = -\frac{\partial E}{\partial x_a^{(i)}}.$$

The force on the conforming node b due to E is

$$\begin{aligned}\tilde{f}_b^{(i)} &= -\frac{\partial E}{\partial \tilde{x}_b^{(i)}} = -\sum_a \frac{\partial E}{\partial x_a^{(i)}} \frac{\partial x_a^{(i)}}{\partial \tilde{x}_b^{(i)}} \\ &= \sum_a f_a^{(i)} \frac{\partial}{\partial \tilde{x}_b^{(i)}} \left(\sum_d W_{ad} \tilde{x}_d^{(i)} \right) \\ &= \sum_a f_a^{(i)} \sum_d W_{ad} \delta_{bd} = \sum_a W_{ab} f_a^{(i)}.\end{aligned}$$

Therefore we distribute forces at the primary nodes to the conforming nodes using $\boldsymbol{W}^T$.

4 Corotational Linear Elasticity

We follow the approach of [McAdams et al. 2011] to corotational linear elasticity and summarize some relevant results here. Using their optimized singular value decomposition (SVD) algorithm, we decompose the deformation gradient as $\boldsymbol{F} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$, where $\boldsymbol{U}$ and $\boldsymbol{V}$ are rotation matrices and $\boldsymbol{\Sigma}$ is diagonal. From the SVD, we form the polar decomposition $\boldsymbol{F} = \boldsymbol{R}\boldsymbol{S}$, where $\boldsymbol{R} = \boldsymbol{U}\boldsymbol{V}^T$ is a rotation matrix and $\boldsymbol{S} = \boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{V}^T$ is a symmetric matrix.

4.1 Stabilized Energy and Forces

The total deformation energy is found by integrating the energy density function Ψ over the domain. We use an efficient stabilized one-point quadrature which splits $\Psi = \Psi_\Delta + \Psi_{\text{aux}}$ into two parts,

$$\begin{aligned}\Psi_\Delta(\boldsymbol{F}) &= \mu \|\boldsymbol{F}\|_F^2 \\ \Psi_{\text{aux}}(\boldsymbol{F}) &= -2\mu \text{tr}(\boldsymbol{S}) + \mu \|\boldsymbol{I}\|_F^2 + \frac{\lambda}{2} \text{tr}^2(\boldsymbol{S} - \boldsymbol{I}),\end{aligned}$$

where μ and λ are the Lamé parameters. The "auxiliary" part Ψ_{aux} is integrated with a one-point quadrature, while a different rule is used for the "Laplacian" part Ψ_Δ.

This induces a splitting on the 1st Piola-Kirchhoff stress tensor $\boldsymbol{P} \triangleq \partial\Psi/\partial\boldsymbol{F} = \boldsymbol{P}_\Delta + \boldsymbol{P}_{\text{aux}}$,

$$\begin{aligned}\boldsymbol{P}_\Delta(\boldsymbol{F}) &= 2\mu\boldsymbol{F} \\ \boldsymbol{P}_{\text{aux}}(\boldsymbol{F}) &= (-2\mu + \lambda \text{tr}(\boldsymbol{S} - \boldsymbol{I}))\, \boldsymbol{R}.\end{aligned}$$

The deformation energy $E^e = E^e_\Delta + E^e_{\text{aux}}$ of a single element is found by integrating the energy density over the element,

$$\begin{aligned}E^e_\Delta &= V_e \int_{[0,1]^3} \Psi_\Delta(\boldsymbol{F}(\boldsymbol{\xi}))\, \mathrm{d}\boldsymbol{\xi} \\ E^e_{\text{aux}} &= V_e \int_{[0,1]^3} \Psi_{\text{aux}}(\boldsymbol{F}(\boldsymbol{\xi}))\, \mathrm{d}\boldsymbol{\xi}.\end{aligned}$$

4.1.1 Auxiliary Part

We approximate the integral for E^e_{aux} using a single quadrature point at the center of the element, $\boldsymbol{\xi}_e = (1/2, 1/2, 1/2)^T$. This gives $E^e_{\text{aux}} \approx V_e \Psi_{\text{aux}}(\boldsymbol{F}^e)$, where $\boldsymbol{F}^e = \boldsymbol{F}(\boldsymbol{\xi}_e)$.

To compute $\boldsymbol{F}^e$ from (2) , we define $\boldsymbol{G}^e$ to be the 3×8 matrix of unmodified shape function derivatives evaluated at the center of the element

$$G^e_{ia} = \left.\frac{\partial N_a}{\partial X^{(i)}}\right|_{\boldsymbol{\xi}_e} = \frac{1}{h} \left.\frac{\partial N_a}{\partial \xi^{(i)}}\right|_{\boldsymbol{\xi}_e}.$$

This is a different way to arrive at the same value for $\boldsymbol{G}^e$ given by [McAdams et al. 2011]. Our interpretation of $\boldsymbol{G}^e$ in terms of shape functions allows us to rigorously extend their method to octrees.

The auxiliary part of the force on the primary node a due to element e is

$$f^{(i)}_{\text{aux}\,a} = -\frac{\partial E^e_{\text{aux}}}{\partial x_a^{(i)}} = -V_e [\boldsymbol{P}_{\text{aux}}(\boldsymbol{F}^e) \boldsymbol{G}^e]_{ia}. \tag{3}$$

4.1.2 Laplacian Part

The Laplacian part of the force on the primary node a due to element e is linear in $\boldsymbol{x}$

$$f^{(i)}_{\Delta a} = -\frac{\partial E_{e\Delta}}{\partial x_a^{(i)}} = -\sum_c \mathring{K}_{\Delta ac} x_c^{(i)}, \tag{4}$$

where

$$\mathring{K}_{\Delta ac} = \frac{2\mu V_e}{h^2} \sum_k \left(\int_{[0,1]^3} \frac{\partial N_a}{\partial \xi^{(k)}} \frac{\partial N_c}{\partial \xi^{(k)}}\, \mathrm{d}\boldsymbol{\xi} \right). \tag{5}$$

We use a stabilized quadrature scheme to approximate the integrals. We observe that $\partial N_a/\partial\xi^{(k)}$ and $\partial N_c/\partial\xi^{(k)}$ are bilinear polynomials that are constant in $\xi^{(k)}$. We use quadrature points with equal weights of $1/4$ in the centers of the 4 edges of the cubic element where $\xi^{(k)}$ varies and the other two coordinates are fixed at either 0 or 1. $\partial N_a/\partial\xi^{(k)}$ is nonzero along only 1 of these 4 edges, and its value is ± 1. The same holds for $\partial N_c/\partial\xi^{(k)}$. There are only two combinations of a and c for which they are both nonzero. For the case $a = c$, their product is 1, and for the other it is -1. As a result, $\mathring{\boldsymbol{K}}_\Delta$ is proportional to the Laplacian matrix for the element e, justifying the terminology "Laplacian part". Instead of forming the matrix, we compute $\boldsymbol{f}_\Delta$ directly as the Laplacian of $\boldsymbol{x}$.

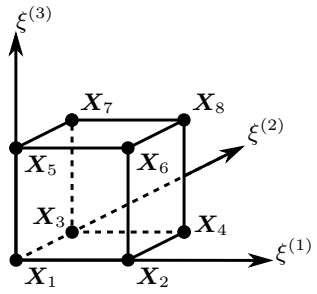


Figure 7: *Labeling of the nodes of a hexahedral element in Morton order.*

4.2 Force Differentials

We solve either the quasistatic elasticity problem or dynamics using implicit integration. In either case, we solve the nonlinear system of equations by generating a series of linearizations using Newton's method. The stiffness matrix $\boldsymbol{K}$ is

$$K_{ac}^{(i)(j)} = -\frac{\partial f_a^{(i)}}{\partial x_c^{(j)}}$$

The force differential $\delta \boldsymbol{f}$ induced by the displacement $\delta \boldsymbol{x}$ is

$$\delta f_a^{(i)} = \sum_{cj} \frac{\partial f_a^{(i)}}{\partial x_c^{(j)}} \delta x_c^{(j)} = -\sum_{cj} K_{ac}^{(i)(j)} \delta x_c^{(j)}.$$

We compute force differentials using a matrix-free method that avoids forming the stiffness matrix $\boldsymbol{K}$.

For the auxiliary part of the force, we take the differential of (3)

$$\delta f_{\text{aux}\,a}^{(j)} = -V_e [\delta \boldsymbol{P}_{\text{aux}}(\boldsymbol{F}^e, \delta \boldsymbol{F}^e) \boldsymbol{G}^e]_{ja},$$

where

$$\delta \boldsymbol{P}_{\text{aux}}(\boldsymbol{F}, \delta \boldsymbol{F}) = \lambda \text{tr}(\boldsymbol{R}^T \delta \boldsymbol{F}) \boldsymbol{R} + \{\lambda \text{tr}(\boldsymbol{S} - \boldsymbol{I}) - 2\mu\} \, \delta \boldsymbol{R}.$$

As in [McAdams et al. 2011], we use the exact differential of the rotation, which is more robust than the warped stiffness formulation of corotational linear elasticity. This is particularly important for the large deformations typical in character skinning applications. The differential of the rotation is

$$\delta \boldsymbol{R} = \boldsymbol{R} \left[\boldsymbol{\mathcal{E}} \left((\text{tr}(\boldsymbol{S}) \boldsymbol{I} - \boldsymbol{S})^{-1} \left(\boldsymbol{\mathcal{E}} : \left(\boldsymbol{R}^T \delta \boldsymbol{F} \right) \right) \right) \right].$$

The Laplacian part of the force $\boldsymbol{f}_\Delta$ is proportional to the Laplacian of $\boldsymbol{x}$. Similarly, we compute $\delta \boldsymbol{f}_\Delta$ by evaluating the Laplacian of $\delta \boldsymbol{x}$.

5 Linear Octrees

Octrees are commonly represented with a pointer-based tree data structure in which the root of the tree has pointers to its 8 children, which in turn have pointers to their children, until reaching the finest octants at the leaves of the tree. In contrast, we use a linear octree [Sundar et al. 2007]. In this representation, only an array of the leaf octants is stored. The intermediate levels of the tree are not stored, and there are no pointers to children or parents. This extra information is either not needed or can be inferred. We use incomplete linear octrees, which means that empty octants are not stored.

We conceptually discretize space using a uniform background grid that represents the finest level of the octree. The local primary node number 1 (figure 7) of an octant is called its "anchor". An octant is uniquely identified by the integer coordinates of its anchor node and its height in the octree.

The same octant data structure represents both an element and its anchor node, with two exceptions. If an element's anchor node is hanging, then the octant represents only an element, and not a conforming node. Conversely, there are some nodes on the boundary that are not the anchor of any element.

Each of the 3 integer coordinates of the anchor is stored in 2 bytes, and the height in 1 byte. This allows a resolution of up to 65,536 octants in each dimension and 256 levels in the octree, which is more than sufficient for character simulation. We store an additional byte in each octant that contains flags. Two of these bits indicate if an octant represents an element, a node or both. Thus the total octant data structure fits in 8 bytes, and a single array of octants is used to represent both the elements and nodes. This representation is very compact compared to a pointer-based octree.

The array of octants is stored in the order of a space-filling Z-shaped curve by sorting the octants by the "Morton code" of their anchor. The Morton code can be constructed by interleaving the bits of the coordinates of the anchor in the order $z_n y_n x_n \ldots z_0 y_0 x_0$. However, it is possible to efficiently compare coordinates according to their Morton code without actually performing this interleaving operation, by using a bitwise exclusive-or [Chan 2002]. Other operations such as calculating parents, children and neighbors of octants can also be implemented efficiently with arithmetic and bitwise logic and shift operations [Burstedde et al. 2011].

We store additional information in separate arrays outside of the octant data structure. An array of 1 byte per octant contains flags indicating whether each of the 8 primary nodes of the octant is hanging. An array of 8 integers per octant contains the indices of the element's 8 conforming nodes in the array of octants. For octants that represent boundary nodes instead of elements, these indices have a different meaning. Instead, we store the index of each element that contains the boundary node. The total number of octants is limited only by the range of the indices, which is over 4×10^9 for a 4-byte unsigned integer. Since so many octants would exhaust the memory of a typical workstation, this is not a limitation in practice. Additional per-node and per-element data, such as deformed positions, deformation gradients, velocities, forces, masses and Lamé parameters are stored in their own arrays.

6 Octree Construction

The input to our simulator is triangle meshes for the elastic objects to be simulated and the bones that they are constrained to. For each triangle, we generate a list of octants at the finest level of the octree that overlap the triangle. We then sort this array of octants in Morton order. We complete the octree in a bottom-up manner in parallel as in [Sundar et al. 2008] by iterating over the array. For each pair of fine octants, we generate the coarsest possible list of octants that fills the space between them in Morton order. After the initial octree is constructed, we 2:1 balance it.

We classify each octant as inside or outside of the elastic objects. All fine octants generated during rasterization are marked as inside. We mark the remaining octants as inside if their center is inside the triangle mesh. We discard the exterior octants, leaving an incomplete linear octree.

We then add the boundary nodes to the octree. For each octant, we search for each of its primary nodes in the array of octants. Since the array of octants is sorted in Morton order, this can be done efficiently using a binary search. If the primary node is not found, it may be a hanging node. We then search for a neighbor octant one level coarser. If it is still not found, then the node is on the boundary and a new octant is added to the octree to represent the boundary node. After all of the boundary nodes have been added, the array of octants is sorted a final time.

After the octree is fully constructed, we determine whether the anchor node of each octant is hanging, again by performing searches for neighbors. We compute the global indices of the conforming nodes of each element with more searches.

7 Geometric Multigrid Preconditioner

At each step of Newton's method, we solve the linearized system using geometric multigrid as a preconditioner to CG. Describing a particular multigrid method requires specifying the cycle type, smoother type, number of smoothing iterations, restriction and prolongation operators, coarse grid operator and the solver on the coarsest level. We use one multigrid V-Cycle, with one presmoothing and one postsmoothing iteration on each level. We use damped Jacobi for smoothing and also as the solver on the coarsest level, because it parallelizes well. We typically use a damping coefficient of 0.3. We use linear interpolation for the prolongation operator. Our restriction operator is the transpose of the prolongation operator. We form the coarse grid operator using a direct coarse grid discretization, by re-discretizing the problem on a coarsened octree.

CG requires the system matrix to be symmetric positive definite. The stiffness matrix $\boldsymbol{K}$ is the Hessian of the energy, so it is symmetric and will be positive definite at the value of $\boldsymbol{x}$ that minimizes the energy. However, during Newton's method, we must evaluate force differentials at other values of $\boldsymbol{x}$, where $\boldsymbol{K}$ may not be positive definite. In order to use CG, we modify $\boldsymbol{K}$ to make it positive definite. This corresponds to a modified Newton method. We use the "indefiniteness correction" algorithm of [McAdams et al. 2011] for computing $\delta \boldsymbol{P}_{\text{aux}}$ (algorithm 1). This algorithm enforces the positive definiteness of the stiffness matrix without explicitly forming it, allowing our method to remain matrix-free. In order to be a valid preconditioner to CG, our multigrid method must be a symmetric positive definite linear operator. Conditions under which this holds are given in [Tatebe 1993], and our choices of smoother, restriction and prolongation operators and coarse grid solver fulfill these conditions. Below we describe several aspects of our multigrid method in more detail.

7.1 Octree Coarsening

We generate a coarse octree from a fine octree by looping over all of the fine elements. For each fine octant, we compute its parent octant. If the only descendants of the parent present in the octree are its immediate children, we replace the children with the parent. Because we use incomplete octrees, not all 8 of the children may be present. This results in the domain growing as the octree is coarsened. Our coarsening is more aggressive than methods which only coarsen the leaves of the octree at each multigrid level.

Even if the fine octree is 2:1 balanced, the coarse octree may not be and must be balanced again. Balancing may re-introduce some finer octants that were coarsened. Due to the way the domain grows during coarsening, some of the finer octants generated during balancing may not be present in the finer octree. These are removed from the coarse octree after balancing.

Algorithm 1 Computation of the stress differential corresponding to the auxiliary energy term Ψ_{aux}. Fixed to guarantee definiteness.

procedure COMPUTELAUX(input $\boldsymbol{\Sigma}$, $\boldsymbol{V}$, μ, λ)
  $\boldsymbol{L}_{D\text{aux}} \leftarrow \{\lambda \text{tr}(\boldsymbol{\Sigma} - \boldsymbol{I}) - 2\mu\} \{\text{tr}(\boldsymbol{\Sigma})\boldsymbol{I} - \boldsymbol{\Sigma}\}^{-1}$
  Clamp diagonal elements of $\boldsymbol{L}_{D\text{aux}}$ to a minimum value of $(-\mu)$ ▷ Term Ψ_Δ will boost this eigenvalue by μ
  $\boldsymbol{L}_{\text{aux}} \leftarrow \boldsymbol{V}\boldsymbol{L}_{D\text{aux}}\boldsymbol{V}^T$
  return $\boldsymbol{L}_{\text{aux}}$
end procedure
procedure DPAUXDEFINITEFIX(input $\delta\boldsymbol{F}$, $\boldsymbol{R}$, $\boldsymbol{L}_{\text{aux}}$)
  $\delta\hat{\boldsymbol{F}}_{\text{sym}} \leftarrow \text{SYMMETRICPART}(\boldsymbol{R}^T\delta\boldsymbol{F})$
  $\delta\hat{\boldsymbol{F}}_{\text{skew}} \leftarrow \text{SKEWSYMMETRICPART}(\boldsymbol{R}^T\delta\boldsymbol{F})$
  $\delta\hat{\boldsymbol{P}}_{\text{sym,aux}} \leftarrow \lambda \text{tr}(\delta\hat{\boldsymbol{F}}_{\text{sym}})\boldsymbol{I}$
  $\delta\hat{\boldsymbol{P}}_{\text{skew,aux}} \leftarrow \boldsymbol{\mathcal{E}}\left\{\boldsymbol{L}_{\text{aux}}\left(\boldsymbol{\mathcal{E}} : \delta\hat{\boldsymbol{F}}_{\text{skew}}\right)\right\}$
  $\delta\boldsymbol{P}_{\text{aux}} \leftarrow \boldsymbol{R}\left(\delta\hat{\boldsymbol{P}}_{\text{sym,aux}} + \delta\hat{\boldsymbol{P}}_{\text{skew,aux}}\right)$
  return $\delta\boldsymbol{P}_{\text{aux}}$
end procedure

7.2 Restriction and Prolongation

Our restriction and prolongation operators are the transpose of one another. We precompute the weights. We store an array of integers that maps each fine node to a coarse element that contains it. We store 8 weights per fine node, which are the weights of the fine node with respect to the conforming nodes of the coarse element that contains it. We build these arrays by looping over the arrays of coarse and fine octants simultaneously, taking advantage of the fact that they are sorted in Morton order.

For each coarse element, we loop through all of the fine elements that are contained in it. For each fine element, if the anchor node is not hanging, we compute its weights. For every boundary node of the fine element, we compute its weights if this is the first fine element that contains it.

In order to compute the weights for a fine node, we first compute its trilinear coordinates $\boldsymbol{\xi} \in [0, 1]^3$ with respect to the primary nodes of the coarse element. We then convert these to weights with respect to the conforming nodes of the coarse element by using $\boldsymbol{W}^T$ on the coarse octree.

For both restriction and prolongation, we always loop over the fine nodes. This ensures that the operations are the transpose of one another. It is also simpler than looping over the coarse elements, because each fine node is embedded in exactly one coarse element, while each coarse element may have many embedded fine nodes.

7.3 Coarse Grid Operator

We form the coarse grid operator using a direct coarse grid discretization, rather than Galerkin coarsening. In addition to coarsening the octree as described above, we must also coarsen all of the information necessary to be able to compute force differentials on the coarsened octree. To coarsen the Lamé parameters, we simply average the Lamé parameters of the 8 child elements, substituting 0 for any missing children. To coarsen the nodal masses, we use the same weights as the restriction operator.

For a hyperelastic material, the energy density function Ψ does not depend directly on the deformed positions, but is a function of the strain only. Therefore, as observed in [McAdams et al. 2011], we do not need to coarsen the positions, but can coarsen only the deformation gradient $\boldsymbol{F}$. This is important because, due to our use of an incomplete octree, the domain grows as it is coarsened. Coarsening

the deformed nodal positions would require extrapolating the positions outside of the original domain, which McAdams et al. found to be unstable. They proposed coarsening the deformation gradient using a weighted average. We have found that a simple average works just as well. We coarsen the deformation gradient by averaging the deformation gradient of all of the *existing* children. This is in contrast to coarsening the Lamé parameters, where we always average over all 8 children, substituting 0 for missing children. Once we have the coarsened material properties and deformation gradient, we can compute force differentials on the coarse octrees in a matrix-free fashion in exactly the same way as on the finest octree.

7.4 The Diagonal of the Stiffness Matrix

Since we never form the stiffness matrix explicitly, the diagonal of the stiffness matrix needed for damped Jacobi smoothing is not readily available. For corotational linear elasticity, the stiffness matrix changes every Newton step and can not be precomputed. Nonetheless, its diagonal part can be computed efficiently. Details of how to compute the diagonal of the stiffness matrix for a uniform grid discretization were given in the technical notes accompanying [McAdams et al. 2011]. Below we present an extension of this computation to octrees. We follow the same general approach, but several simplifying assumptions do not apply to octrees due to the need to modify the shape functions for hanging nodes. For a detailed derivation of these results, see [Milne 2015].

We compute the diagonal entry $\tilde{d} = -\partial \tilde{f}_b^{(i)} / \partial \tilde{x}_b^{(i)}$ of the element stiffness matrix corresponding to component i of the conforming node b. It is not possible to compute the diagonal of the stiffness matrix with respect to the conforming nodes using only the diagonal of the stiffness matrix with respect to the primary nodes. The off-diagonal entries of the latter contribute the the diagonal entries of the former. Therefore, we derive equations for the diagonal of the stiffness matrix in terms of the modified shape functions. We split $\tilde{d} = \tilde{d}_\Delta + \tilde{d}_{\text{aux}}$ into the Laplacian and auxiliary parts and calculate them separately.

7.4.1 Auxiliary Part

We define $\tilde{\boldsymbol{G}}^e$ to be the 3×8 matrix of modified shape function derivatives evaluated at the center of the element

$$\tilde{G}^e_{ib} = \left.\frac{\partial \tilde{N}_b}{\partial X^{(i)}}\right|_{\boldsymbol{\xi}_e} = \sum_a W_{ab} \left.\frac{\partial N_a}{\partial X^{(i)}}\right|_{\boldsymbol{\xi}_e} = \sum_a W_{ab} G^e_{ia}.$$

Let $\tilde{\boldsymbol{g}}$ be column b of $\tilde{\boldsymbol{G}}^e$ and $\boldsymbol{r}^T$ be row i of $\boldsymbol{R}^e$. Let $\boldsymbol{L}_{\text{aux}}$ be the matrix computed in algorithm 1 for the indefiniteness correction of $\delta \boldsymbol{P}_{\text{aux}}$. Then

$$\tilde{d}_{\text{aux}} = V_e \left(\lambda (\boldsymbol{r}^T \tilde{\boldsymbol{g}})^2 + (\tilde{\boldsymbol{g}} \times \boldsymbol{r})^T \boldsymbol{L}_{\text{aux}} (\tilde{\boldsymbol{g}} \times \boldsymbol{r}) \right).$$

7.4.2 Laplacian Part

For a uniform grid, the diagonal entries of the Laplacian part of the stiffness matrix are all equal to a constant value. For octrees, these entries are no longer all equal. However, they are constant and can be precomputed. Writing (4) and (5) in terms of the conforming nodes instead of the primary nodes leads to

$$\tilde{d}_\Delta = \frac{\partial^2 E^e_\Delta}{\partial \tilde{x}_b^{(i)2}} = \frac{2\mu V_e}{h^2} \sum_k \left(\int_{[0,1]^3} \left(\frac{\partial \tilde{N}_b}{\partial \xi^{(k)}} \right)^2 \mathrm{d}\boldsymbol{\xi} \right).$$

As in section 4.1.2, we approximate the integral by averaging the integrand along the 4 edges of the cubic element where $\xi^{(k)}$ varies and the other two coordinates are fixed at either 0 or 1. Let Γ_k denote this set of 4 edges. For each edge $\gamma \in \Gamma_k$, denote the primary node where $\xi^{(k)} = 0$ by a_0, and the primary node where $\xi^{(k)} = 1$ by a_1. Let $\boldsymbol{\xi}_0$ be the parameter of a_0 and $\boldsymbol{\xi}_1$ be the parameter of a_1. Since $\partial \tilde{N}_b / \partial \xi^{(k)}$ is constant along each edge in Γ_k, we can calculate it as

$$\frac{\partial \tilde{N}_b}{\partial \xi^{(k)}} = \tilde{N}_b(\boldsymbol{\xi}_1) - \tilde{N}_b(\boldsymbol{\xi}_0) = W_{a_1 b} - W_{a_0 b}.$$

Then

$$\tilde{d}_\Delta = \frac{2\mu V_e}{4h^2} \sum_k \sum_{\gamma \in \Gamma_k} (W_{a_1 b} - W_{a_0 b})^2 .$$

When there are no hanging nodes, this agrees with the value in [McAdams et al. 2011] for uniform grids, $d_\Delta = 3\mu V_e / 2h^2$.

8 Constraints and Collisions

We attach the flesh to the bones using zero rest length springs embedded in the elements. Let $\boldsymbol{\xi}_p$ be the parameter of the constrained point, $\boldsymbol{x}_0$ be the target position for the constraint and $\boldsymbol{x}_p$ be the current deformed position of the point given by (1). The constraint force on primary node a is

$$f_a^{(i)} = -k(\boldsymbol{x}_p - \boldsymbol{x}_0) N_a(\boldsymbol{\xi}_p),$$

where k is the spring constant. The force differential is

$$\delta f_a^{(i)} = -k \delta \boldsymbol{x}_p N_a(\boldsymbol{\xi}_p).$$

The diagonal of the stiffness matrix for the conforming node b is

$$\tilde{d} = -\frac{\partial \tilde{f}_b^{(i)}}{\partial \tilde{x}_b^{(i)}} = k(\tilde{N}_b(\boldsymbol{\xi}_p))^2.$$

We also use zero rest length springs for collision repulsion forces, but separate the force into normal and tangential components. We use a lower spring constant in the tangential direction to allow some sliding and simulate friction. We perform all collision and self-collision detection using triangle meshes, accelerated with a bounding volume hierarchy. For collisions with objects, we query the closest point on the collision object for each deformed vertex $\boldsymbol{x}_p$ of the embedded triangle mesh. This yields a target position $\boldsymbol{x}_0$ and surface normal $\boldsymbol{n}$ on the collision object.

We compute the self-intersection curves on the triangle mesh to determine regions of self-intersection. For each deformed vertex $\boldsymbol{x}_{p1}$ in the self-intersecting region, we shoot a ray in the positive and negative normal directions to find a target triangle. Using the barycentric coordinates of the hit point and the undeformed triangle mesh, we compute the octant and parameter $\boldsymbol{\xi}_{p2}$ of the target point in material space. We then compute the deformed position of the target point $\boldsymbol{x}_{p2}$ using (1). Due to differences in the number of degrees of freedom of the triangle mesh and the octree, and barycentric interpolation versus interpolation using the element shape functions, this may yield a slightly different point than the original one computed by ray intersection. We have found that it is important to use $\boldsymbol{x}_{p2}$ for consistency. We compute a normal for the self-collision by $\boldsymbol{n} = (\boldsymbol{x}_{p2} - \boldsymbol{x}_{p1}) / \|\boldsymbol{x}_{p2} - \boldsymbol{x}_{p1}\|$.

9 Parallel Implementation

We have parallelized our solver with a multithreaded shared-memory CPU implementation. We divide our linear octree into

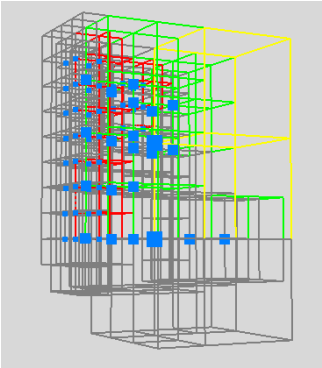


Figure 8: *One thread block. Elements in the block are color-coded by size in red, green and yellow. Nodes in the block are shown in blue. Ghost octants are in gray.*

Algorithm 2 Computation of the Force Differential in Parallel

```
1: procedure ForceDifferential(input octree, δx̃_octree;
                                output δf̃_octree)
2:     for block in octree parallel do
3:         for element e in BlockIterator(block) do
               ▷ includes ghost cells
4:             δx_e ← GatherFromOctree(octree, e, δx̃_octree)
5:             δf_e ← ElementForceDifferential(δx_e)
6:             ScatterToOctree(octree, e, block, δf_e, δf̃_octree)
                   ▷ writes only to nodes in block
7:         end for
8:     end for
9: end procedure
```

blocks of consecutive octants which are processed in parallel. Since the octree is sorted in Morton order, consecutive octants in the array will tend to be adjacent spatially. When processing a block, the thread only writes to the anchor nodes of the octants within that block. However, the thread will need to read the values of nodes that are outside of the block. We call an octant that contributes to the nodes in the block, but whose anchor does not belong to the block, a "ghost cell". We compute the list of ghost cells for each block during initialization. Figure 8 shows a thread block with ghost cells. Choosing the block size involves balancing competing considerations. A smaller block size facilitates load balancing, while a larger block size reduces the number of duplicate computations due to the ghost cells. We chose a block size of 64, but have not tuned this to determine the optimal block size.

We use a block iterator that iterates through octants in the block and the ghost cells. Since the ghost cells usually have smaller indices than the octants in the block, we sort the ghost cells when we generate them and iterate over the ghost cells first, followed by the octants in the block. This order has proven to be most efficient.

An example of using this block iterator is algorithm 2 for computing force differentials. The input is an array of displacements at the conforming nodes of the octree, and the output is an array of force differentials. We iterate over the blocks in parallel. Within each block, we iterate over its ghost cells and octants. The GatherFromOctree procedure computes the displacements at the 8 primary nodes of an element from the displacements at the conforming nodes of the octree, implementing the operation we represent as multiplication by $\boldsymbol{W}$. The ElementForceDifferential procedure computes all of the force differentials (elastic, constraints, collisions, etc.) for a single element in a matrix-free manner as described in sections 4.2 and 8. Finally, ScatterToOctree distributes the force differentials at the primary nodes of the element to the conforming nodes of the octree, implementing the operation we represent as multiplication by $\boldsymbol{W}^T$. This procedure only writes to the nodes in the current block.

Algorithm 3 Restriction of the Residual in Parallel

```
1: procedure RestrictResidual(input coarse_octree, r_fine;
                               output r_coarse)
       ▷ cb_fnl = coarse_block_to_fine_node_list
       ▷ fn_ce = fine_node_to_coarse_element
       ▷ w = fine_node_to_coarse_element_weights
2:     for block in coarse_octree parallel do
3:         ClearBlock(block, r_coarse)
4:         for conforming node b_fine in cb_fnl[block] do
5:             element e_coarse ← fn_ce[b_fine]
6:             for conforming node b_coarse in element e_coarse do
7:                 if b_coarse in block then
8:                     r_coarse[b_coarse] += w[b_fine][b_coarse] * r_fine[b_fine]
9:                 end if
10:            end for
11:        end for
12:    end for
13: end procedure
```

Computing forces in parallel is implemented in a similar way to force differentials. We also parallelize the damped Jacobi smoother and the computation of the diagonal of the stiffness matrix using the block iterator. Many other computations, such as the deformation gradient and collision detection, are trivially parallelizable.

We compute the restriction of the residual in parallel as shown in algorithm 3. As described in section 7.2, we precompute and store the arrays *fine_node_to_coarse_element* and *fine_node_to_coarse_element_weights*. To facilitate parallelization, for each coarse block we precompute the list of fine nodes that contribute to it, which are stored in the array *coarse_block_to_fine_node_list*.

For the prolongation operation, we simply iterate over the fine conforming nodes in parallel and compute the contributions of the coarse conforming nodes to each fine conforming node using *fine_node_to_coarse_element* and *fine_node_to_coarse_element_weights*. In this direction there is no need to use thread blocks since all of the coarse conforming nodes that contribute to a given fine conforming node can be found on a single coarse element.

We coarsen the deformation gradient once per Newton iteration as described in section 7.3 and shown in algorithm 4. The deformation gradient is computed at the center of the element and is therefore an element-based quantity instead of a node-based quantity like the forces and displacements. Since the octrees are sorted in Morton order, all of the fine elements contained in a coarse element are contiguous, and we can loop over the two arrays of octants simultaneously. We precompute the first index of the fine elements for each coarse block and store them in *coarse_block_to_fine_element_start*.

10 Results

We present results for two examples that demonstrate the applicability of our method to different types of scenarios. The hippo example (figure 1a, 1b) is representative of a production-level character in an animated feature film. We collected performance data for the hippo on an animated walk cycle and also ran our solver on an animation performance test. The Spike example (figure 1c, 1d)

Algorithm 4 Coarsening of the Deformation Gradient in Parallel

```
1:  procedure CoarsenDeformationGradient(
                        input coarse_octree, F_fine; output F_coarse)
              ▷ cb_fes = coarse_block_to_fine_element_start
              ▷ fe_ce = fine_element_to_coarse_element
2:      for block in coarse_octree parallel do
3:          element e_fine ← cb_fes[block]
4:          for element e_coarse in block do
              ▷ excludes ghost cells
5:              w = 0, F = 0
6:              while e_fine < cb_fes[block + 1] do
7:                  if fe_ce[e_fine] > e_coarse then
8:                      break
9:                  end if
10:                 F += F_fine[e_fine]
11:                 w += 1
12:                 e_fine ← next fine element
13:             end while
14:             F_coarse[e_coarse] = F/w
15:         end for
16:     end for
17: end procedure
```

contains several scales of features in order to demonstrate the effectiveness of our method on domains with complex geometry that would be infeasible to simulate with a uniform grid. Figure 9 shows a close-up view of the fine elements around the boundary. The head is constrained to a smaller sphere embedded in the center, which is animated up and down and rotated side to side, before a cylindrical collision object passes through the spikes.

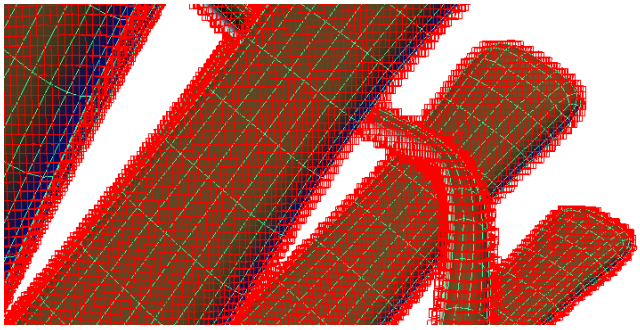


Figure 9: *Close-up view of Spike showing the fine elements of size 0.25 around the boundary. The fine elements are able to resolve the complex geometry.*

10.1 Memory Usage

We ran both examples with a range of cell sizes to compare the memory usage of our octree-based method to a uniform grid implemented as a 3D array data structure. We compared an octree with a given finest cell size to a uniform grid of the same cell size. Both the octree and the uniform grid have the same size cell around the boundary, but with the octree, the cells become coarser towards the interior of the object. We measured the total number of elements and memory usage on a workstation with 192GB of RAM. Results are plotted in figure 10 as a function of the inverse of the cell size.

For the uniform grid, the growth of both the number of elements and memory usage is cubic as the resolution increases. As mentioned previously, when a character is embedded in a uniform grid, many of the grid cells are empty, which wastes memory. Combined with

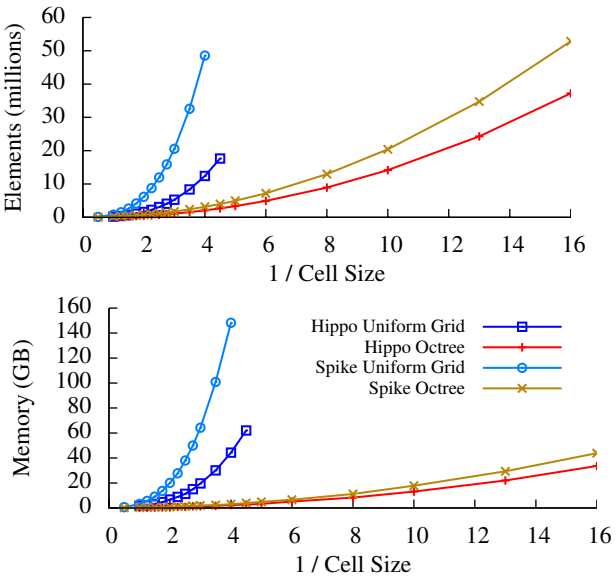


Figure 10: *Number of elements and memory usage versus inverse cell size. Memory was measured as virtual memory usage for the solver after simulating one frame. For both the number of elements and memory usage, the growth is cubic for the uniform grid, and quadratic for the octree.*

the cubic growth in the number of elements, this places a severe limitation on how small of cells can be used before exhausting the available memory.

For the octree, the growth of the number of elements and memory usage is only quadratic as the resolution increases. This is because we refine around the surface of the character, rather than throughout the entire volume. In addition, we omit the empty cells in our linear octree data structure. This allows us to achieve a very small finest cell size without approaching the limits of available memory.

10.2 Performance

We measured the performance of our implementation using 40 threads on a 40 core 2.20GHz Intel Xeon E5-2698 v4 workstation with 64GB of RAM. Our code was vectorized using AVX instructions. Figure 11 plots the average simulation time per frame as a function of the inverse cell size for the hippo on an octree and a uniform grid. For the largest cell size of 1.0, the uniform grid had 209k elements and simulated in an average of 0.367 seconds per frame. The octree with a finest cell size of 1.0 had 101k elements and simulated in an average of 0.352 seconds per frame. For moderate to large cell sizes, the overhead of the octree is compensated for by the reduction in the number of elements, and the total simulation times for the octree are slightly faster than the uniform grid.

However, as the cell size decreases, the octree simulations become significantly faster than the uniform grid. As shown in figure 12, the increase in simulation times is linear in the number of elements for both the uniform grid and octree. This is a consequence of our use of a multigrid method, which has linear scaling in the number of unknowns in the ideal case. Therefore, the simulation times inherit the cubic and quadratic growth rates in the number of elements, as can be observed in figure 11.

Due to extreme memory usage, the smallest possible cell size for the uniform grid was 1/3.5, corresponding to 8.4 million elements and an average simulation time of 10.8 seconds per frame. At the

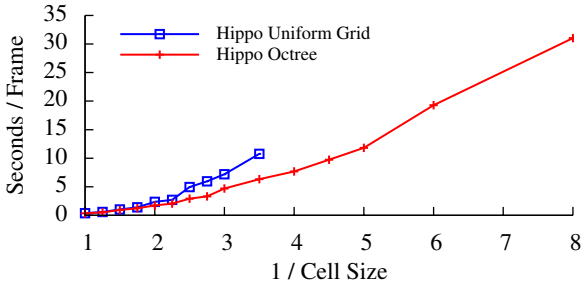


Figure 11: *Average simulation times per frame versus inverse cell size for the hippo example. The growth is cubic for the uniform grid, and quadratic for the octree.*

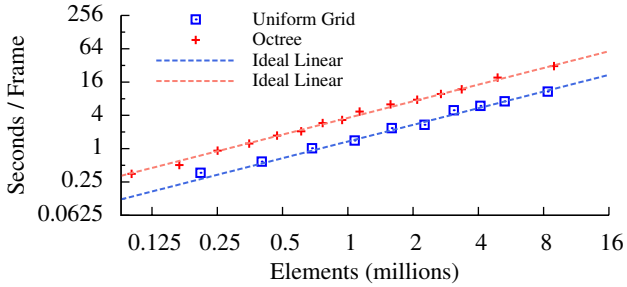


Figure 12: *Average simulation times per frame versus number of elements for the hippo example. For both the uniform grid and octree, the growth is linear in the number of elements. The overhead incurred by the octree is offset by the fewer number of elements for the same cell size, compared to the uniform grid.*

same finest cell size, the octree had only 1.6 million elements and an average simulation time of 6.3 seconds per frame. It was possible to run octree simulations with even smaller cell sizes, down to a finest cell size of 1/8, corresponding to 8.9 million elements and an average simulation time of 31.0 seconds per frame.

We ran the Spike example on an octree with a smallest cell size of 1/4 (shown in figures 1c and 9), resulting in 3.1 millions elements. It was not feasible to run this simulation using a uniform grid, and larger cell sizes were insufficient to capture the thin features of the geometry. Due to the highly flexible and unconstrained motion of this example, it was necessary to solve to higher tolerance than required for a more typical character such as the hippo. Insufficient convergence is manifested as artificial damping on the motion of the thin features. We set the maximum number of Newton steps to 20 for this example, compared to a more typical value of 10. Over 119 frames of simulation, this example averaged 11.4 Newton steps per frame. We also set the maximum number of multigrid preconditioned CG iterations per Newton step to 20, compared to a typical value of 3. With these settings the average simulation time per frame was 26.2 seconds. With a maximum of 10 Newton steps per frame and 3 MGPCG iterations per Newton step, the average simulation time per frame was 8.8 seconds, but some numerical damping could be observed.

10.3 Parallel Scalability

To measure the parallel scalability of our method, we ran the hippo example with a finest cell size of 1/2.5 (shown in figure 1a), resulting in 765k elements, on 1, 2, 4, 8, 16, 24, 32 and 40 threads. Results are shown in figure 13, compared to the ideal linear scaling. The parallel efficiency is above 0.90 for up to 16 threads, and we still achieve a parallel efficiency of 0.75 for 40 threads.

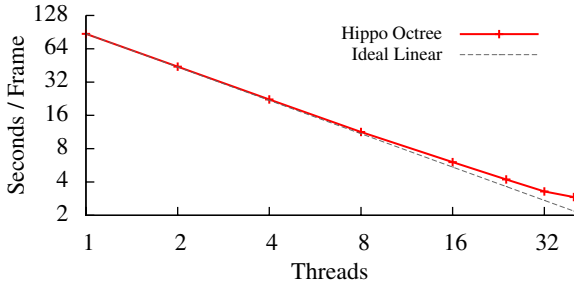


Figure 13: *Average simulation times per frame versus number of threads for the hippo example on an octree. We achieve near linear scaling in the number of threads.*

10.4 Convergence

To measure the convergence of our method, we performed a quasistatic simulation of a cube of 256 units per side using an octree with a finest cell size of 1 unit per side along the boundary. The 8 corner elements of the cube were constrained, and the position of each unconstrained node was randomly displaced by an amount in the range $[-100, 100]$ in each dimension. We measured the convergence back to the rest state. In order to isolate the convergence of the linear solver, only one Newton iteration was used. We compared three different linear solvers: conjugate gradient (CG), multigrid (MG) and multigrid-preconditioned conjugate gradient (MGPCG). For both MG and MGPCG, we used 7 multigrid levels and a damped-Jacobi smoother with a damping coefficient of 0.857 , with one smoothing iteration per multigrid level. The coarsest multigrid level was solved with 16 damped-Jacobi iterations. For MGPCG, one V-cycle of MG was used as a preconditioner each CG iteration.

Figure 14 shows the reduction in the max norm of the error versus the time in seconds. In our application, this is the most appropriate measurement of convergence, as a large error in any component of the position will be visually apparent, even if the overall error as measured by the energy norm or l^2 norm is small. CG without a preconditioner is seen to converge very slowly, and in fact the max norm of the error actually increases for the first few iterations, even while the energy norm of the error is decreasing. MGPCG outperforms MG, especially as the number of iterations increases. As shown in figure 15, MGPCG converges visually in fewer iterations than MG. Each MGPCG iteration is slightly more expensive than a MG iteration. But the added expense of the MGPCG iteration is more than compensated for by the improved convergence rate, and the MGPCG method still converges fastest when measured in wall-clock time.

11 Discussion

In our method the octree is constructed in an initialization step and does not change during the course of the simulation. Adapting the octree in response to the simulation would increase accuracy in areas of high deformation, for example due to collisions. We also do not support topology changes during the simulation such as cutting or tearing. In our current implementation, such changes would require an expensive re-initialization of the multigrid hierarchy and other data structures. Incrementally updating these data structures without sacrificing the efficiency of our method is an area of future work.

The derivation of our method is intimately tied to the equations

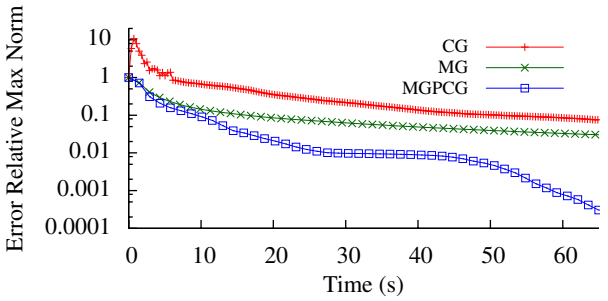
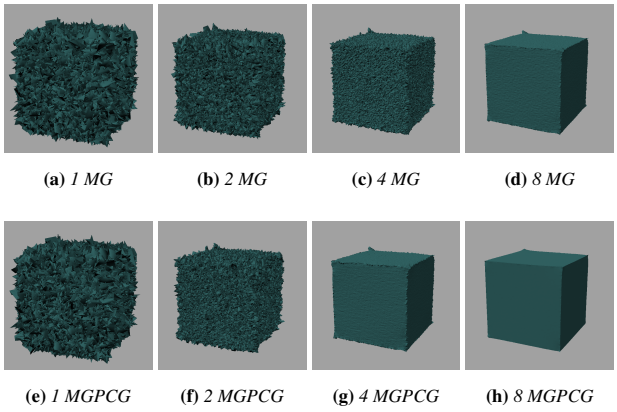


Figure 14: *Relative reduction of the max norm of the error versus time in seconds for CG, MG and MGPCG. Each marker is one iteration of the method. Times are reported for a single thread on a 16 core 2.70 GHz Intel Xeon E5-2680 workstation with 128GB of RAM using code not optimized for SIMD vectorization.*



(a) *1 MG* **(b)** *2 MG* **(c)** *4 MG* **(d)** *8 MG*

(e) *1 MGPCG* **(f)** *2 MGPCG* **(g)** *4 MGPCG* **(h)** *8 MGPCG*

Figure 15: *Convergence of multigrid linear solver with a random initial guess by number and type of iterations. The corners of the cube are constrained.*

of corotational linear elasticity. Another area of future work is to extend the method to other nonlinear constitutive models, including incompressible and anisotropic materials. Anisotropic materials are important for modeling the direction of muscle fibers. But the convergence of multigrid methods suffers in the presence of anisotropy unless more sophisticated smoothers, such as line or plane smoothers, are used. Using more expensive smoothers may be detrimental to efficiency and parallelism.

In computer animation, it is often desirable to change the rest state of the object over time in order to achieve artistic effects. This also occurs in the real world, where the rest state may change due to plasticity. However, the assumption that the rest state of each element is a regular hexahedron is deeply built into our method and is the basis for much of its efficiency. Supporting these types of effects is an interesting challenge.

We have focused on a parallel shared-memory CPU implementation. Our results show the high parallel scalability of our method, but the parallelism is limited by the number of cores on a single workstation. Therefore, it may be worthwhile to pursue a distributed-memory or GPU implementation to achieve even greater parallelism.

Our method combines the benefits of an adaptive discretization of corotational linear elasticity with the efficiency of methods previously only applicable to regular grids. This enables the practical simulation of highly complex and detailed geometry. We are excited about both the potential and challenges of the integration of physical simulation into animation workflows that is made possible by highly efficient solvers.

Acknowledgments

The authors would like to thank Nicholas Burkard, Heather Pritchett and Nicklas Puetz for help with the hippo example.

References

BOUAZIZ, S., MARTIN, S., LIU, T., KAVAN, L., AND PAULY, M. 2014. Projective dynamics: Fusing constraint projections for fast simulation. *ACM Trans. Graph. 33*, 4 (July), 154:1–154:11.

BRAESS, D. 1986. On the combination of the multigrid method and conjugate gradients. In *Multigrid Methods II*, W. Hackbusch and U. Trottenberg, Eds., vol. 1228 of *Lecture Notes in Math.* Springer Berlin Heidelberg, 52–64.

BURSTEDDE, C., WILCOX, L., AND GHATTAS, O. 2011. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM J. Comput. 33*, 3, 1103–1133.

CHAN, T. M. 2002. Closest-point problems simplified on the RAM. In *Proc. Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, SODA '02, 472–473.

CLUTTERBUCK, S., AND JACOBS, J. 2010. A physically based approach to virtual character deformations. In *ACM SIGGRAPH 2010 Talks*.

COMER, S., BUCK, J., AND CRISWELL, B. 2015. Under the scalpel - ILM's digital flesh workflows. In *ACM SIGGRAPH 2015 Talks*, 10:1–10:1.

DICK, C., GEORGII, J., AND WESTERMANN, R. 2011. A hexahedral multigrid approach for simulating cuts in deformable objects. *IEEE Trans. Vis. Comput. Graphics 17*, 11 (Nov), 1663–1675.

FLAIG, C., AND ARBENZ, P. 2012. A highly scalable matrix-free multigrid solver for μFE analysis based on a pointer-less octree. In *Large-Scale Scientific Computing*, I. Lirkov, S. Margenov, and J. Waniewski, Eds., vol. 7116 of *Lecture Notes in Comput. Sci.* Springer Berlin Heidelberg, 498–506.

FRATARCANGELI, M., TIBALDO, V., AND PELLACINI, F. 2016. Vivace: A practical Gauss-Seidel method for stable soft body dynamics. *ACM Trans. Graph. 35*, 6 (Nov.), 214:1–214:9.

GEORGII, J., AND WESTERMANN, R. 2010. A streaming approach for sparse matrix products and its application in Galerkin multigrid methods. *Electron. Trans. Numer. Anal. 37*, 263–275.

IRVING, G., KAUTZMAN, R., CAMERON, G., AND CHONG, J. 2008. Simulating the devolved: Finite elements on WALL•E. In *ACM SIGGRAPH 2008 Talks*, 54:1–54:1.

ISAAC, T., BURSTEDDE, C., AND GHATTAS, O. 2012. Low-cost parallel algorithms for 2:1 octree balance. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th Int.*, 426–437.

KAUTZMAN, R., CHONG, J., AND COLEMAN, P. 2012. Stable, art-directable skin and flesh using biphasic materials. In *ACM SIGGRAPH 2012 Talks*.

KAUTZMAN, R., WISE, B., YU, M., KARLSSON, P., HESSLER, M., AND WONG, A. 2016. Finding Hank: Or how to sim an octopus. In *ACM SIGGRAPH 2016 Talks*, 61:1–61:2.

KAVAN, L., COLLINS, S., ŽÁRA, J., AND O'SULLIVAN, C. 2008. Geometric skinning with approximate dual quaternion blending. *ACM Trans. Graph. 27*, 4 (Nov.), 105:1–105:23.

LEWIS, J. P., CORDNER, M., AND FONG, N. 2000. Pose space deformation: A unified approach to shape interpolation and skeleton-driven deformation. In *Proc. SIGGRAPH '00*, Annual Conference Series, 165–172.

LIN, W.-C., ZAFAR, N. B., NG, E., AND ZHOU, J. 2016. Pyramid coordinates for deformation with collision handling. In *ACM SIGGRAPH 2016 Talks*, 33:1–33:2.

MCADAMS, A., SIFAKIS, E., AND TERAN, J. 2010. A parallel multigrid Poisson solver for fluids simulation on large grids. In *Proc. 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Eurographics Association, SCA '10, 65–74.

MCADAMS, A., ZHU, Y., SELLE, A., EMPEY, M., TAMSTORF, R., TERAN, J., AND SIFAKIS, E. 2011. Efficient elasticity for character skinning with contact and collisions. *ACM Trans. Graph. 30*, 4 (July), 37:1–37:12.

MCLAUGHLIN, T., CUTLER, L., AND COLEMAN, D. 2011. Character rigging, deformations, and simulations in film and game production. In *ACM SIGGRAPH 2011 Courses*, 5:1–5:18.

MILNE, A., TAMSTORF, R., STOMAKHIN, A., AND MCLAUGHLIN, M., 2016. Geometric multigrid on incomplete linear octrees for simulating deformable animated characters. Patent US 20160292902 A1, Oct. 6.

MILNE, A., MCLAUGHLIN, M., TAMSTORF, R., STOMAKHIN, A., BURKARD, N., COUNSELL, M., CANAL, J., KOMOROWSKI, D., AND GOLDBERG, E. 2016. Flesh, flab, and fascia simulation on Zootopia. In *ACM SIGGRAPH 2016 Talks*, 34:1–34:2.

MILNE, A. 2015. *A Geometric Multigrid Preconditioner on Incomplete Linear Octrees for Simulating Deformable Animated Characters*. Master's thesis, California State University, Northridge.

MITCHELL, N., DOESCHER, M., AND SIFAKIS, E. 2016. A macroblock optimization for grid-based nonlinear elasticity. In *Pro. ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Eurographics Association, SCA '16, 11–19.

PATTERSON, T., MITCHELL, N., AND SIFAKIS, E. 2012. Simulation of complex nonlinear elastic bodies using lattice deformers. *ACM Trans. Graph. 31*, 6 (Nov.), 197:1–197:10.

SAMPATH, R. S., AND BIROS, G. 2010. A parallel geometric multigrid method for finite elements on octree meshes. *SIAM J. Sci. Comput. 32*, 3, 1361–1392.

SAMPATH, R. S., ADAVANI, S. S., SUNDAR, H., LASHUK, I., AND BIROS, G. 2008. Dendro: Parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees. In *Proc. 2008 ACM/IEEE Conference on Supercomputing*, IEEE Press, SC '08, 18:1–18:12.

SCHUTZ, V., GIACOPPO, P., AND COOPER, C. 2016. Warcraft's Durotan: Hero, complex. In *ACM SIGGRAPH 2016 Talks*, 44:1–44:2.

SEILER, M., SPILLMANN, J., AND HARDERS, M. 2010. A threefold representation for the adaptive simulation of embedded deformable objects in contact. *J. of WSCG 18*, 1-3 (Feb.), 89–96.

SUNDAR, H., SAMPATH, R. S., ADAVANI, S. S., DAVATZIKOS, C., AND BIROS, G. 2007. Low-constant parallel algorithms for finite element simulations using linear octrees. In *Proc. 2007 ACM/IEEE Conference on Supercomputing*, 25:1–25:12.

SUNDAR, H., SAMPATH, R. S., AND BIROS, G. 2008. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM J. Sci. Comput. 30*, 5, 2675–2708.

SUNDAR, H., STADLER, G., AND BIROS, G. 2014. Comparison of multigrid algorithms for high-order continuous finite element discretizations. arXiv:1402.5938v1, Feb.

TATEBE, O. 1993. The multigrid preconditioned conjugate gradient method. In *Sixth Copper Mountain Conference on Multigrid Methods*, NASA, Hampton, VA, N. D. Melson, T. A. Manteuffel, and S. F. McCormick, Eds., vol. CP 3224, 621–634.

TORRES, R., RODRÍGUEZ, A., ESPADERO, J. M., AND OTADUY, M. A. 2016. High-resolution interaction with corotational coarsening models. *ACM Trans. Graph. 35*, 6 (Nov.), 211:1–211:11.

WANG, H., AND YANG, Y. 2016. Descent methods for elastic body simulation on the GPU. *ACM Trans. Graph. 35*, 6 (Nov.), 212:1–212:10.

WANG, W. 2000. Special bilinear quadrilateral elements for locally refined finite element grids. *SIAM J. Sci. Comput. 22*, 6, 2029–2050 (electronic).