Shortest Path to Boundary for Self-Intersecting Meshes

HE CHEN, University of Utah, USA
ELIE DIAZ, University of Utah, USA
CEM YUKSEL, University of Utah & Roblox, USA

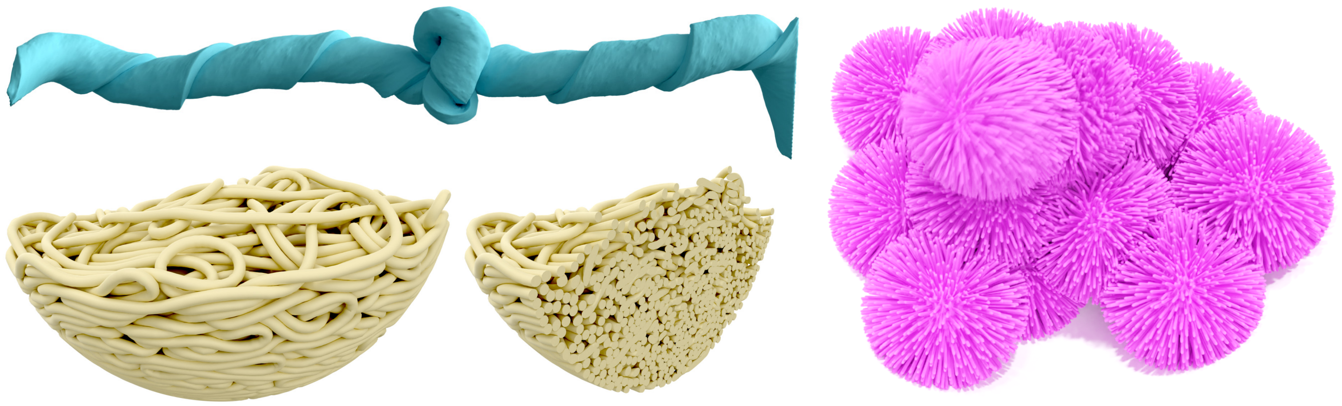


Fig. 1. *Example simulation results involving complex self-collision scenarios, generated using our method with XPBD [Macklin et al. 2016].*

We introduce a method for efficiently computing the exact shortest path to the boundary of a mesh from a given internal point in the presence of self-intersections. We provide a formal definition of shortest boundary paths for self-intersecting objects and present a robust algorithm for computing the actual shortest boundary path. The resulting method offers an effective solution for collision and self-collision handling while simulating deformable volumetric objects, using fast simulation techniques that provide no guarantees on collision resolution. Our evaluation includes complex self-collision scenarios with a large number of active contacts, showing that our method can successfully handle them by introducing a relatively minor computational overhead.

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

Additional Key Words and Phrases: Collision response, Computational geometry, geodesics, shortest path

ACM Reference Format:
He Chen, Elie Diaz, and Cem Yuksel. 2023. Shortest Path to Boundary for Self-Intersecting Meshes. *ACM Trans. Graph.* 42, 4 (August 2023), 15 pages. https://doi.org/10.1145/3592136

Authors' addresses: He Chen, ankachan92@gmail.com, University of Utah, Salt Lake City, UT, USA; Elie Diaz, elie.diaz@utah.edu, University of Utah, Salt Lake City, UT, USA; Cem Yuksel, cem@cemyuksel.com, University of Utah & Roblox, Salt Lake City, UT, USA.

1 INTRODUCTION

Self-intersecting meshes, though they are often highly undesirable, are commonplace in computer graphics. They can appear due to the limitations of modeling techniques, animation methods, or manual editing operations. Even physics-based simulations with self-collision handling are not immune to self-intersections, as most of them cannot guarantee an intersection-free state.

Notwithstanding the amount of work on self-intersection handling within physics-based simulations, it still remains a challenge in most cases. *Continuous collision detection* techniques [Li et al. 2020] require starting with and maintaining an intersection-free state; therefore, they must be used with computationally-expensive methods that can always resolve all self-intersections and they fail when combined with cheaper techniques that are unable to do so. Methods that split an object into pieces [Macklin et al. 2020] turn the self-intersection problem into intersections of these separate pieces, entirely avoiding the self-intersection problem, and they fail to resolve self-intersections within a piece. Methods that solve self-intersections using an intersection-free pose [McAdams et al. 2011] not only require such a pose, but also become inaccurate as the objects deform and fail with sufficiently large deformations and deep penetrations. Therefore, none of these methods provides a robust and general solution for self-intersections.

In this paper, we present a method that robustly and efficiently finds the *exact shortest internal path* of a point inside a mesh to its boundary, even in the presence of self-intersections and some inverted elements. We achieve this by introducing a precise definition of the shortest path to the mesh boundary, including points that are both on the boundary and inside the mesh at the same time, an unavoidable condition with self-intersections. Our approach works with tetrahedral meshes in 3D (with boundaries forming triangular meshes) and triangular meshes in 2D (with polyline boundaries).

We demonstrate that one important application of our method is solving arbitrary self-intersections after they appear in deformable simulations, allowing the use of cheaper integration techniques that do not guarantee complete collision resolution.

Our method is based on the realizations that (1) the shortest path must be fully contained within the geodesic embedding of the mesh and (2) it must be a line segment under Euclidean metrics. Based on these, given a candidate boundary point, our method quickly checks if the line segment to this point is contained within the mesh. Combined with a spatial acceleration structure, we can efficiently find and test the candidate closest boundary points until the shortest path is determined. We also describe a fast and robust tetrahedral traversal algorithm that avoids infinite loops, needed for checking if a path is within the mesh. Furthermore, we propose an additional acceleration that can quickly eliminate candidate boundary points based on local geometry without the need for checking their paths.

One application of our method is resolving intersections between separate objects and self-intersections alike within a fast physics-based simulation system that cannot guarantee intersection-free states. It can be used alone or as a backup for continuous collision detection to handle cases when the simulation system fails to resolve a previously-detected collision. In either case, we achieve a robust collision handling method that can solve extremely challenging cases, involving numerous deep self-intersections, using a fast simulation system that does not provide any guarantees about collision resolution. As a result, we can simulate highly complex scenarios with a large number of self-collisions and rest-in-contact conditions, as shown in Figure 1.

2 RELATED WORK

One important application of our method is collision handling (Section 2.1), though we actually introduce a method for certain types of geodesic distances and paths (Section 2.2). A core part of our method is tetrahedral ray traversal (Section 2.3). In this section, we overview the prior in these areas and briefly present how our approach compares to them.

2.1 Collision Handling

Collision handling is directly related to how they are detected, which can be done using either *continuous collision detection* (CCD) or *discrete collision detection* (DCD).

Starting with an intersection-free state, CCD can detect the first time of contact between elements [Canny 1986], but requires maintaining an intersection-free state. Through the use of a strong barrier function, *incremental potential contact* (IPC) [Li et al. 2020] provides guaranteed collision resolution combined with a CCD-aware line search. This idea was later extended to rigid [Ferguson et al. 2021] and almost rigid bodies [Lan et al. 2022a]. Incorporating projective dynamics into IPC offers performance improvement [Lan et al. 2022b], but resolving all collisions still remains expensive. Even when the simulation system is able to resolve all collisions, CCD itself can fail due to numerical issues, in which case, it can no longer help with resolving the collision, resulting in objects linking together [Wang et al. 2021].

In contrast, DCD allows the simulation framework to start and recover from a state with existing intersections. DCD detects collisions at a single point in time, after they happen. That is why, extra computation is needed to determine how to resolve the collisions.

Collisions can be resolved by minimizing the penetration volume [Allard et al. 2010; Wang et al. 2012] or by applying constraints [Bouaziz et al. 2014; Macklin et al. 2016; Müller et al. 2007; Verschoor and Jalba 2019], penalty forces [Belytschko and Neal 1991; Ding and Schroeder 2019; Drumwright 2007; Huněk 1993], or impulses [Kavan 2003; Mirtich and Canny 1995; O'Sullivan and Dingliana 1999] that involve computing the *penetration depth*, the minimum translational distance to resolve the penetration [Hirota et al. 2000; Platt and Barr 1988; Terzopoulos et al. 1987]. The exact penetration depth can be computed using analytical methods based on geometric information of polygonal meshes [Baraff 1994; Cameron 1997; Hahn 1988; Moore and Wilhelms 1988], or it can be approximated using a volumetric mesh [Fisher and Lin 2001a], mesh partitioning [Redon and Lin 2006], tracing rays [Hermann et al. 2008], or solving an optimization problem [Je et al. 2012]. Heidelberger et al. [2004] proposed a consistent penetration depth by propagating penetration depth through the volumetric mesh. These methods, however, struggle with handling self-intersections. Starting with a self-intersecting shape, Li and Barbič [2018] proposed a method to separate the overlapping parts and create a bounding case mesh that represents the underlying geometry to allow "un-glued" simulation.

Using a *signed distance fields* (SDF) is a more popular alternative for recent methods. They can be defined either on a volumetric mesh [Fisher and Lin 2001a] or a regular grid [Gascuel 1993; Koschier et al. 2017; Macklin et al. 2020]. Once built, both the penetration depth and the shortest path to the surface can be directly queried from the volumetric data structure. This provides an efficient solution at run time as long as the SDF does not need updating, though the returned penetration depth and shortest path are approximations (formed by interpolating pre-computed values). Also, the SDF is not well defined when there are self-intersections, as they cannot represent immersion, so it must be built using an intersection-free pose.

For handling self-intersections, SDFs of an intersection-free pose can be used [McAdams et al. 2011]. This can provide sufficient accuracy for handling minor deformations, but quickly becomes inaccurate with large deformations and deep penetrations. Using a deformable embedding helps [Macklin et al. 2020], but requires splitting the object into pieces [Fisher and Lin 2001a,b; Macklin et al. 2020; McAdams et al. 2011; Teng et al. 2014]. An alternative approach is bifurcating the SDF nodes during construction when a volumetric overlap, which can be formed by self-intersection, is detected [Mitchell et al. 2015]. These solutions entirely circumvent the self-intersection problem by only considering intersections of separate pieces and self-intersections within a piece are ignored. Such approaches are particularly problematic with complex models and in cases when determining where to split is unclear ahead of time, since the splitting or bifurcation is usually pre-computed and expensive to update at run time. Also, the closest boundary point found within a piece is not necessarily the actual one for the entire mesh, as it might be contained in a separate piece. Even for cases they can handle with sufficient accuracy, SDFs have a significant pre-computation and storage cost.

In comparison, our solution can find the *exact* penetration depth for models with arbitrary complexity and the accurate shortest path to the boundary regardless of the type or severity of self-intersections. In addition, we do not require costly pre-computations or volumetric storage.

2.2 Geodesic Path and Distances

Following the categorization of Crane et al. [2020], our method falls into the category of *multiple source geodesic distance/shortest path* (MSGD/MSSP) problems. Actually, the problem we solve is a special case of MSSP, where the set of sources is the collection of all the boundary points of the mesh. Also, ours is an exact polygonal method that can compute global geodesic paths. MMP algorithm [Mitchell et al. 1987] is the first practical algorithm that can compute geodesic path between any two points on a polygonal surface. Succeeding methods [Chen and Han 1990; Liu 2013; Surazhsky et al. 2005; Xin and Wang 2009] focus on optimizing its computation time and memory requirements. Yet, all of these method only aim at solving the *single source geodesic distance/shortest path* (SSGD/SSSP) problems. For solving the *all-pairs geodesic distances/shortest paths* (APGD/APSP) problem, a vertex graph that encodes the minimal geodesic distances between all pairs of vertices on the mesh can be built [Balasubramanian et al. 2008]. These methods are general enough for handling 2D manifolds in 3D, but they do not offer an efficient solution for our MSSP problem. Our solution for MSSP, however, is limited to planar (2D, triangular) or volumetric (3D, tetrahedral) meshes, where we can rely on Euclidean metrics.

2.3 Tetrahedral Ray Traversal

For handling tetrahedral meshes in 3D, our method uses a topological ray traversal. Tetrahedral ray traversal has been used in volumetric rendering [Marmitt and Slusallek 2006; Parker et al. 2005; Şahıstan et al. 2021]. Methods that improve their computational cost include using scalar triple products [Lagae and Dutré 2008] and Plucker coordinates [Maria et al. 2017]. More recently, Aman et al. [2022] introduced a highly-efficient dimension reduction approach.

A common problem with tetrahedral ray traversal is that numerical inaccuracies can lead to infinite loops when a ray passes near an edge or vertex. Many rendering problems can safely terminate when an infinite loop is detected. In our case, however, we must detect and resolve such cases, because failing to do so would result in returning an incorrect shortest path, which can have catastrophic effects in simulation. Therefore, we introduce a robust variant of tetrahedral ray traversal.

3 SHORTEST PATH TO BOUNDARY

A typical solution for resolving intersections (detected via DCD) is finding the closest boundary point for each intersecting point and then applying corresponding forces/constraints along the line segment toward this point, i.e. the shortest path to boundary. The length of this path is the penetration depth.

When two separate objects intersect, finding the closest boundary point is a trivial problem: it is the closest boundary point on the other object. In the case of self-intersections, however, even the definition of the shortest path to boundary is somewhat ambiguous.

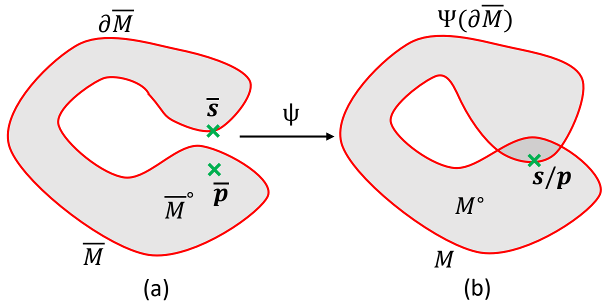


Fig. 2. *Illustrations of the notations. (a) Notations on the undeformed pose. (b) Notations on the deformed model. The image of the undeformed pose boundary $\Psi(\partial\overline{M})$ is marked as the red curve.*

Consider a point on the boundary and also inside the object due to self-intersections. Since this point is already on the boundary, its Euclidean closest boundary point would be itself. Yet, this information is not helpful for resolving the self-intersection.

In this section, we provide a formal definition of the shortest path to boundary based on the geodesic path of the object in the presence of self-intersections (Section 3.1). Then, we present an efficient algorithm to compute it for triangular/tetrahedral meshes in 2D/3D, respectively, (Section 3.2). We also describe how to handle meshes that contain some inverted elements, (Section 3.5). The resulting method provides a robust solution for handling self-collisions that can be used with various simulation methods and collision resolution techniques (using forces or constraints).

3.1 Shortest Path to Boundary

Consider a self-intersecting model M, such that a boundary point $\mathbf{s}$ coincides with an internal point $\mathbf{p}$. Figure 2b shows a 2D illustration, though the concepts we describe here apply to 3D (and higher dimensions) as well. In this case, $\mathbf{s}$ and $\mathbf{p}$ have the same geometric positions, but topologically they are different points. In fact, to fix the self-intersection, we need to apply a force/constraint that would move $\mathbf{s}$ along $\mathbf{p}$'s geodesic shortest path to boundary.

To provide a formal definition of this geodesic shortest path, we consider a self-intersection-free form of this model as $\overline{M}$ which we call undeformed pose, and a deformation Ψ that maps all points in $\overline{M}$ to its current shape M, such that $M = \Psi(\overline{M})$. Note that our algorithm (explained in Section 3.2) does not actually need computing $\overline{M}$ or Ψ. For any point $\bar{\mathbf{p}}$ in $\overline{M}$, we represent its image under Ψ as $\mathbf{p} \in M$, such that $\mathbf{p} = \Psi(\bar{\mathbf{p}})$. In the following, we assume that $\overline{M}$ is a path-connected (i.e. a single piece) manifold, though the concepts below can be trivially extended to models with multiple separate pieces.

To cause self-intersection, Ψ should not be injective. In this case, Ψ is an immersion of $\overline{M}$ but not embedding, meaning multiple points from $\overline{M}$ are mapped to the same position $\mathbf{p}$ inside M. To differentiate such points that coincide in M, we label them using their unique

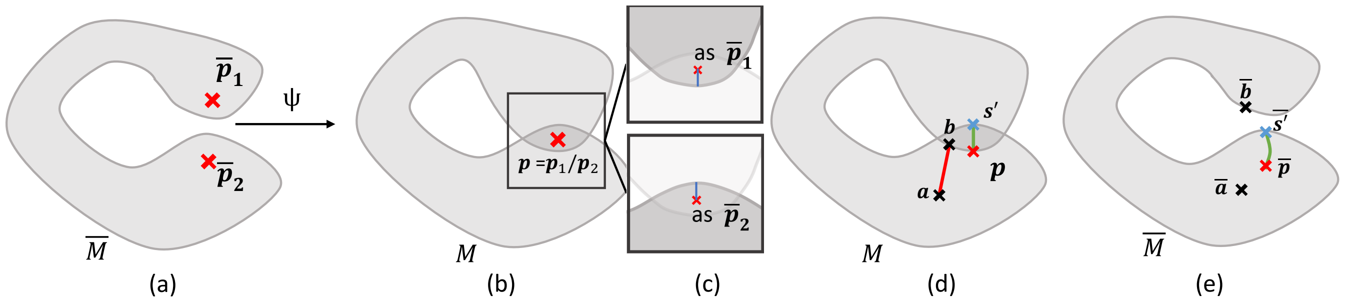


Fig. 3. *(a) An intersection-free pose of the deformable model $\overline{M}$. $\overline{\mathbf{p}}_1, \overline{\mathbf{p}}_2 \in \overline{M}^\circ$. (b) $\overline{M}$'s image under Ψ, where $\overline{\mathbf{p}}_1, \overline{\mathbf{p}}_2$ are mapped to the same point* $\mathbf{p}$. *(c) Treated as different pre-image,* $\mathbf{p}$ *has different shortest paths to the boundary (blue line). (d) Two paths are contained by M. (c) Only* $\mathbf{ps'}$ *is a valid path.*

positions in $\overline{M}$. For simplicity, we say $\mathbf{p}$ *as* $\overline{\mathbf{p}}$, when we are referring $\mathbf{p}$ as the image of $\overline{\mathbf{p}}$.

For simplicity, let us consider non-degenerate Ψ that forms no inversion, i.e. $\det(\nabla\Psi) > 0$. We discuss inversions later in Section 3.5. Note that under this Ψ, the boundary of the undeformed model $\partial\overline{M}$ does not completely overlap with the boundary of the deformed model ∂M, i.e. $\Psi(\partial\overline{M}) \neq \partial M$, see Figure 2b. We use $\overline{M}^\circ$ to denote the set of interior points of $\overline{M}$, such that $\overline{M} = \partial\overline{M} \cup \overline{M}^\circ$.

Let $\mathbf{s}$ as be a point on the boundary, i.e. $\mathbf{s} \in \Psi(\partial\overline{M})$ and we refer to it as an undeformed pose boundary point $\overline{\mathbf{s}}$. For a given point $\mathbf{p}$ (as $\overline{\mathbf{p}}$), we can construct a path $\mathbf{c}(t) : [0, 1] \mapsto M$ as a continuous curve that connects $\mathbf{p} = \mathbf{c}(0)$ to $\mathbf{s} = \mathbf{c}(1)$.

Definition 1 (Valid path). *The path* $\mathbf{c}(t)$ *from* $\mathbf{p}$ *(as* $\overline{\mathbf{p}}$*) to* $\mathbf{s}$ *(as* $\overline{\mathbf{s}}$*) is a* valid path *if there exists a continuous curve* $\overline{\mathbf{c}}(t) : [0, 1] \mapsto \overline{M}$ *such that* $\mathbf{c}(t) = \Psi(\overline{\mathbf{c}}(t))$, $\overline{\mathbf{c}}(0) = \overline{\mathbf{p}}$, $\overline{\mathbf{c}}(1) = \overline{\mathbf{s}}$.

Based on this definition, a *valid path* must be the image of a path that is fully contained within $\overline{M}$, which connects the two points on the undeformed pose we are referring to. Any path that moves outside of $\overline{M}$ is considered an *invalid path*, see Figure 3de. Our goal is to find the shortest valid path from a given point $\mathbf{p}$ (as $\overline{\mathbf{p}}$) to the boundary.

Definition 2 (Shortest path to boundary). *For an interior point* $\mathbf{p}$ *(as* $\overline{\mathbf{p}}$*), the* shortest path to boundary *is the shortest curve* $\mathbf{c}(t)$ *in M that connects* $\mathbf{p}$ *to a boundary point* $\mathbf{s}$ *(as* $\overline{\mathbf{s}}$*) that is a valid path between* $\mathbf{p}$ *and* $\mathbf{s}$.

Definition 3 (Closest boundary point). *For an interior point* $\mathbf{p}$ *(as* $\overline{\mathbf{p}}$*), the* closest boundary point *is the boundary point* $\mathbf{s}$ *(as* $\overline{\mathbf{s}}$*) at the other end of* $\mathbf{p}$*'s shortest path to boundary* $\mathbf{c}(t) = \Psi(\overline{\mathbf{c}}(t))$, *such that* $\mathbf{s} = \mathbf{c}(1)$ *and* $\overline{\mathbf{s}} = \overline{\mathbf{c}}(1)$.

Here we must emphasize that the definition of the shortest path is dependent on the pre-image point we are referring to. For a point located at the overlapping part of M, referring to it as a different point on the undeformed pose may lead to a different shortest path to the boundary (see Figure 3c). Also, this definition is equivalent to the image of $\overline{\mathbf{p}}$'s global geodesic path to boundary in $\overline{M}$ evaluated under the metrics pulled back by Ψ. Thus the shortest path we defined is a special class of geodesics.

To construct an efficient algorithm for finding the shortest path, we rely on two properties:

- First, by definition, the shortest path must be a continuous curve that is fully contained inside undeformed model $\overline{M}$.
- Second, the shortest path (under the Euclidean distance metrics) that connects two points in the deformed model M must be a line segment.

Based on these properties, we can construct and prove the fundamental theorem of our algorithm:

Theorem 1. *For any point* $\mathbf{p} \in M$ *(as* ($\overline{\mathbf{p}}$)*, its shortest path to the boundary is the shortest line segment from* $\mathbf{p}$ *to a boundary point* $\mathbf{s} \in \Psi(\partial\overline{M})$ *(as* $\overline{\mathbf{s}}$*), that is a valid path.*

Here we verbally prove the theorem, we also provide a formal proof in the supplementary document. If the shortest path is not a line segment, we can continuously deform it into a line segment, while keeping the end points fixed. This procedure can induce a deformation on the undeformed pose, which continuously deforms the pre-image of that curve to the pre-image of the line segment, while keeping the end points fixed. This is always achievable because the curve cannot touch the boundary of the undeformed pose during the deformation, otherwise, we will form an even shorter path to the boundary. Thus the line segment is also a valid path.

Based on these properties, our algorithm investigates a set of candidate boundary points $\mathbf{s}$ and checks if the line segment from the interior point $\mathbf{p}$ to $\mathbf{s}$ is a valid path. This is accomplished without having to construct $\overline{M}$ or determine the deformation Ψ by relying on the topological connections of the given discretized model.

3.2 Shortest Path to Boundary for Meshes

In practice, models we are interested in are discretized in a piecewise linear form. These are triangular meshes in 2D and tetrahedral meshes in 3D. We refer to each piecewise linear component as an *element* (i.e. a triangle in 2D and a tetrahedron in 3D) and the one-dimension-lower-simplex shared by two topologically-connected elements as a *face* (i.e. an edge between two triangles in 2D and a triangular face between two tetrahedra in 3D). This discretization makes it easy to test the validity of a given path, without constructing a self-intersection-free $\overline{M}$ or the related deformation Ψ.

We propose the concept of *element traversal* for meshes, as a sequence of topologically connected elements:

Definition 4 (Element traversal). *For a mesh M, and two-point $\mathbf{a} \in e_\mathbf{a}, \mathbf{b} \in e_\mathbf{b}$, we define a element traversal from $\mathbf{a}$ to $\mathbf{b}$ as a list of elements $\mathcal{T}(\mathbf{a}, \mathbf{b}) = (e_0, e_1, e_2, \ldots, e_k)$, where e_i is a element of M, $e_0 = e_\mathbf{a}$, $e_k = e_\mathbf{b}$, and $e_i \cap e_{i+1}$ must be a face.*

Specifically, we call it *tetrahedral traversal* for 3D meshes, and *triangular traversal* for 2D meshes.

Let $\mathbf{c}(t)$ be a line segment from a point $\mathbf{p}$ inside an element $e_\mathbf{p}$ to a boundary point $\mathbf{s}$ of a boundary element $e_\mathbf{s}$ (with a boundary face that contains $\mathbf{s}$). If $\mathbf{c}(t)$ is a valid path, there must be a corresponding piecewise linear path $\overline{\mathbf{c}}(t)$ in $\overline{M}$ from $\overline{\mathbf{p}}$ to $\overline{\mathbf{s}}$ that passes through an element traversal of $\overline{M}$. Actually, an element traversal containing $\mathbf{c}(t)$ is the sufficient and necessary condition for $\mathbf{c}(t)$ being a valid path. Please see the supplementary material for a rigorous proof.

Thus, evaluating whether $\mathbf{c}(t)$ is a valid path, is equivalent to searching for an element traversal from $\overline{\mathbf{s}}$ to $\overline{\mathbf{p}}$, and a piece-wise linear curve $\overline{\mathbf{c}}(t) : I \mapsto \overline{M}$ defined on it, such that $\mathbf{c}(t) = \Psi(\overline{\mathbf{c}}(t))$. Such an element traversal and piece-wise linear curve can be efficiently constructed in M.

Going through the element traversal, $\overline{\mathbf{c}}(t)$ must pass through faces shared by neighoring elements at points $\overline{\mathbf{r}}_i \in e_i \cap e_{i+1}$, where $i = 0, 1, 2, \ldots, k-1$. When Ψ forms no inversion, corresponding face points $\mathbf{r}_i$ must be along the line segment $\mathbf{c}(t)$, i.e. $\mathbf{r}_i = \mathbf{c}(t_i)$ for some $t_i \in [0, 1]$, see Figure 4a. If we can form such an element traversal using the topological connections of the model, we can safely conclude that the path is valid.

This gives us an efficient mechanism for testing the validity of the shortest path from $\mathbf{p}$ to $\mathbf{s}$. Starting from $e_\mathbf{p}$, we trace a ray from $\mathbf{p}$ towards $\mathbf{s}$ and find the first face point $\mathbf{r}_0$. If $\mathbf{r}_0$ is not on the boundary, this face must connect $e_\mathbf{p}$ to a neighboring element e_1. Then, we enter e_1 from $\mathbf{r}_0$ and trace the same ray to find the exit point $\mathbf{r}_1$ on another face. We continue traversing until we reach $e_\mathbf{s}$, in which case we can conclude that this is a valid path, see Figure 4a. This also includes the case $e_\mathbf{p} = e_\mathbf{s}$. If we reach a face point $\mathbf{r}_i$ that is on the boundary (see Figure 4b) or we pass-through $\mathbf{s}$ without entering $e_\mathbf{s}$, $\mathbf{s}$ cannot be the closest boundary point to $\mathbf{p}$.

This process allows us to efficiently test the validity of a path to a given boundary point, but we have infinitely many points on the boundary to test. Fortunately, we are only interested in the shortest path and we can use the theorem below to test only a single point per boundary face.

Theorem 2. *For each interior point $\mathbf{p}$ (as $\overline{\mathbf{p}}$), if its closest boundary point $\mathbf{s}$ (as $\overline{\mathbf{s}}$) is on the boundary face f, $\mathbf{s}$ must also be the Euclidean closest point to $\mathbf{p}$ on f.*

The proof is similar to Theorem 1, which is included in the supplementary document. Based on Theorem 2, we only need to check a single point (the Euclidean closest point) on each boundary face to find the closest boundary point. If we test these boundary points in the order of increasing distance from the interior point $\mathbf{p}$, as soon as we find a valid path to one of them, we can terminate the search by returning it as the closest boundary point. In practice, we use a BVH (bounding volume hierarchy) to test these points, which allows testing them approximately (though not strictly) in the order of increasing distance and, once a valid path is found, quickly skipping the further away bounding boxes.

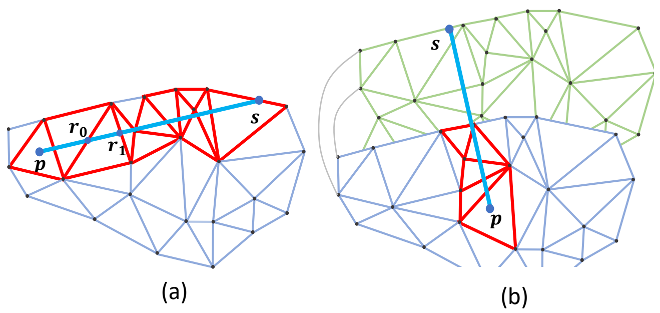


Fig. 4. *(a) An example of a triangular traversal, marked by red triangles. A line segment connecting* **p** *and* **s** *is included in this triangular traversal. (b) An example of a line segment being an invalid path when there are self-intersections, the triangular traversal (marked by the red triangles) stops at the boundary of the mesh but the line segment penetrates the boundary and continues going.*

3.3 Robust Topological Ray Traversal

The process we describe above for testing the validity of the linear path to a candidate boundary point involves traversing a ray through the mesh. This ray traversal is significantly simpler than typical ray traversal algorithms used for rendering with ray tracing. This is because it directly follows the topological connections of the mesh.

At each step, the ray enters an element through one of its faces and must exit from one of its other faces. Therefore, we do not need to rely on an acceleration structure to quickly determine which faces to test ray intersections, as they are directly known from the mesh topology. In fact, we do not need to check each one of the other faces individually, since the ray exits from exactly one of them. Therefore, we can quickly test all possible exit faces together.

For example, Aman et al. [2022] present such a tetrahedral traversal algorithm in 3D. Yet, due to limited numerical precision, this algorithm is prone to forming infinite loops. Such infinite loops are easy to detect and terminate (e.g. using a maximum iteration count), but such premature terminations are entirely unacceptable in our case. This is because incorrectly deciding on the validity of a path would force our algorithm to pick an incorrect shortest path to boundary, which can be arbitrarily far from the correct one. Therefore, the simulation system that relies on this shortest path to boundary can place strong and arbitrarily incorrect forces/constrains in an attempt to resolve the self-intersection.

Our solution for properly resolving such cases that arise from limited numerical precision is three fold:

(1) We allow ray intersections with more than one face by effectively extending the faces using a small tolerance parameter ϵ_i in the intersection test. This forms branching paths when a ray passes between multiple faces and, therefore, intersects (within ϵ_i) with more than one of them.
(2) We keep a list of traversed elements and terminate a branch when the ray enters an element that was previously entered.
(3) We keep a stack containing all the candidate intersecting faces from the intersection test. After a loop is detected, we pick the latest element from it and continue the process.

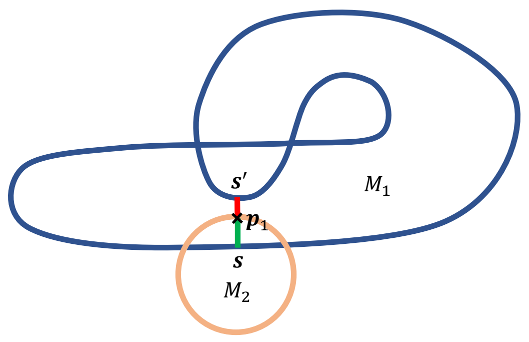


Fig. 5. *An object M_2 intersects with a self-intersecting object M_1. A surface point of M_2 is overlapping with an interior point $\mathbf{p}_1 \in M_1$. $\mathbf{s}$ and $\mathbf{s}'$ are $\mathbf{p}_1$'s closest boundary point by our definition and Euclidean closest boundary point, respectively.*

Please see our supplementary material for the pseudo-code and more detailed explanations of our algorithm.

In practice such branching happens rarely, but solution ensures that we never incorrectly terminate the ray traversal. Note that ϵ_i is a conservative parameter for extending the ray traversal through branching to prevent problems of numerical accuracy issues. It does not introduce any error to the final shortest paths we find. Using an unnecessarily large ϵ_i would only have negative, though mostly imperceptible, performance consequences. We verified this by making the ϵ_i ten times larger, which did not result in a measurable performance difference.

One corner case is when the internal point $\mathbf{p}$ (as $\overline{\mathbf{p}}$) and the boundary point $\mathbf{s}$ (as $\overline{\mathbf{s}}$) coincide, such that $\mathbf{p} = \mathbf{s}$ (within numerical precision). This forms a line segment with zero length and, therefore, does not provide a direction for us to traversal. This happens when testing self-intersections of boundary points, which pick themselves as their first candidate for the closest boundary point. This zero-length line segment cannot be a valid path. Fortunately, since we know we are testing self-intersection for $\mathbf{s}$, when the BVH query returns the boundary face includes $\mathbf{s}$, we can directly reject it.

3.4 Intersections of Different Objects

Although our method is mainly designed for solving self-intersections, it is still needed for handling intersections of different objects when they may have self-intersections as well. As shown in Figure 5, an object M_2 intersects with a self-intersecting object M_1, where a surface point of M_2 is overlapping with an interior point $\mathbf{p}_1 \in M_1$. Simply querying for $\mathbf{p}_1$'s Euclidean closest boundary point in M_1 will give us $\mathbf{s}'$, which does not help resolve the penetration. This is because $\mathbf{p}_1\mathbf{s}'$ is not a valid path between $\mathbf{p}_1$ (as $\overline{\mathbf{p}}_1 \in \overset{\circ}{\overline{M}_1}$) and $\mathbf{s}'_1$ as ($\overline{\mathbf{s}}' \in \partial\overline{M}_1$). What is actually needed is $\mathbf{p}_1$'s shortest path to boundary as $\overline{\mathbf{p}}_1$, which is the same problem as the self-intersection case, a surface point of M_1 is overlapping with an interior point $\mathbf{p}_1 \in M_1$.

3.5 Inverted Elements

Our derivations in Section 3.1 assume that $det(\nabla\Psi) > 0$ everywhere. For a discrete mesh, this would mean no inverted or degenerate elements. Unfortunately, though inverted elements are often highly undesirable, they are not always unavoidable. Fortunately, the algorithm we describe above can be slightly modified to work in the presence of certain types of inverted elements.

If the inverted elements are not a part of the mesh boundary, we can still test the validity of paths by allowing the ray traversal to go backward along the ray. This is because the ray would need to traverse backward within inverted elements. In addition, we cannot simply terminate the traversal once the ray passes through the target point, because an inverted element further down the path may cause backward traversal to reach (or pass through) the target point, see Figure 6b. Therefore, ray traversal must continue until a boundary point is reached. We also need to allow the ray to go behind the starting point, see Figure 6c.

A consequence of this simple modification to our algorithm is that, when we begin from an internal point $\mathbf{p}$ toward a boundary point $\mathbf{s}$, it is unclear if we would reach $\mathbf{s}$ by beginning the traversal toward $\mathbf{s}$ or in the opposite direction. While one may be more likely, both are theoretically possible.

To avoid this decision, in our implementation we start the traversal from the target boundary point $\mathbf{s}$. In this case, there is no ambiguity, since there is only one direction we can traverse along the ray. This also allows using the same traversal routine for the first element and the other elements along the path by always entering an element from a face. Therefore, it is advisable even in the absence of inverted elements.

Nonetheless, our algorithm is not able to handle all possible inverted elements. For example, if the inverted element is on the boundary, as shown in Figure 7, the inversion itself can cause self-intersection. In such a case, a surface point $\mathbf{s}$ is overlapping with an interior point $\mathbf{p}$ (as $\overline{\mathbf{p}}$). Our algorithm will not be able to try to construct a tetrahedral traversal between those two points because we cannot determine a ray direction for a zero-length line segment. Actually, in this case, the very definition of the closest boundary point can be ambiguous.

Our solution is to skip the self-intersection detection of inverted boundary elements. As a result, the only way for us to solve such self-intersections caused by inverted boundary elements is to resolve the inversion itself. Fortunately, inverted elements are undesirable for most simulation scenarios, and they are often easier to fix for boundary elements. Unfortunately, if the inverted boundary elements have global self-intersections with other parts of the mesh, our solution ignores them. Though this does not form a complete solution, because the inverted boundary elements are rare, the other boundary elements surrounding the inverted elements are often enough to solve the global self-intersection.

3.6 Infeasible Region Culling

In a lot of cases, it is possible to determine that a given candidate boundary point $\mathbf{s}$ cannot be the closest boundary point to an interior point $\mathbf{p}$, purely based on the local information about the mesh around $\mathbf{s}$, without performing any ray traversal. For this test we construct a particular region of space, i.e. the *feasible region*, around $\mathbf{s}$. When $\mathbf{p}$ is outside of this region of $\mathbf{s}$, thus in its *infeasible region*, we can safely conclude that $\mathbf{s}$ is not the closest boundary point.

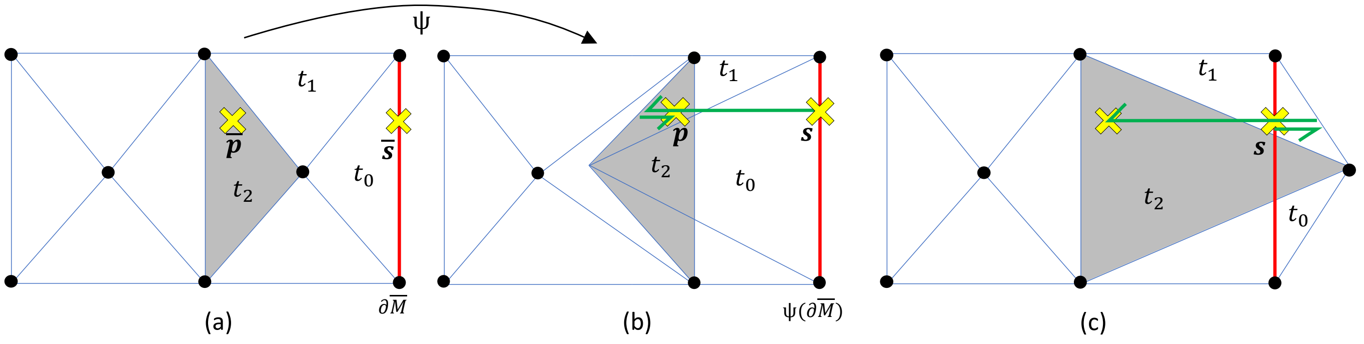


Fig. 6. *(a) A part of the undeformed pose of a triangular mesh $\overline{M}$, which is inversion free. $\overline{\mathbf{p}} \in \overline{M}$, $\overline{\mathbf{s}} \in \partial\overline{M}$. A surface edge is marked with red color. (b) The image of $\overline{M}$ under Ψ, the tetrahedron t_2 (colored with gray), is inverted by Ψ. The green line illustrates $\mathbf{p}$'s global geodesics to the surface, it has a self-overlapping part, which is marked by the two-sided arrow. (c) An interior tetrahedron is inverted and got out of the surface. In this case, the global geodesics to the surface path can go backward.*

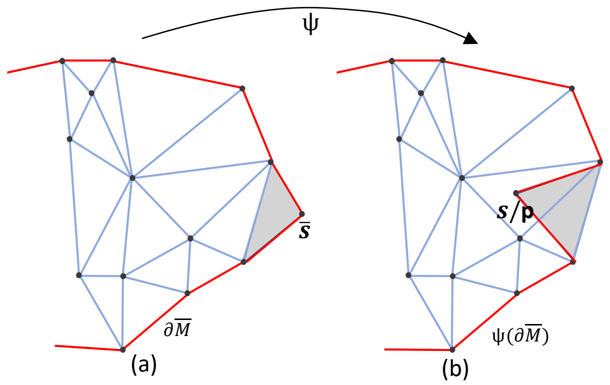


Fig. 7. *(a) A part of the undeformed pose of a triangular mesh $\overline{M}$, which is inversion free. The surface edges are marked with red color. (b) After deformation, a triangle (marked by gray color), is inverted and folded into the interior of the mesh. A deformed surfaces point $\mathbf{s}$ overlaps with the interior point $\mathbf{p}$.*

The construction of this feasible region depends on whether $\mathbf{s}$ is on a vertex, edge, or face.

Vertex Feasible Region. In 2D, when $\mathbf{s}$ is on a vertex, the feasible region is bounded by the two lines passing through the vertex and perpendicular to its two boundary edges, as shown in Figure 8a. For a neighboring boundary edge of $\mathbf{s}$ and its perpendicular line that passes through $\mathbf{s}$, if $\mathbf{p}$ is on the same side of the line as the edge, based on Theorem 2, there must be a closer boundary point on the face. More specifically, for any neighboring boundary vertex $\mathbf{v}_i$ connected to $\mathbf{s}$ by a edge, if the following inequality is true, $\mathbf{p}$ is in the infeasible region:

$$(\mathbf{p} - \mathbf{s}) \cdot (\mathbf{s} - \mathbf{v}_i) < 0 \tag{1}$$

The same inequality holds in 3D for all neighboring boundary vertices $\mathbf{v}_i$ connected to $\mathbf{s}$ by an edge (Figure 8b). The 3D version of the vertex feasible region is actually the space bounded by a group of planes perpendicular to its neighboring edges.

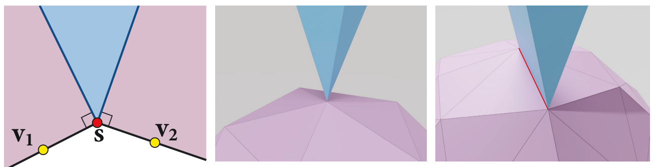


Fig. 8. *The feasible region, shaded in blue, for (a) a boundary vertex in 2D, (b) a boundary vertex in 3D, and (c) a boundary edge in 3D. Note that the 3D meshes in (b) and (c) are observed from the inside.*

Edge Feasible Region. In 3D, when $\mathbf{s}$ is on the edge of a triangle, its feasible region is the intersection of 4 half-spaces defined by four planes: two planes that contain the edge and perpendicular to its two adjacent faces, and two others that are perpendicular to the edge and pass through its two vertices, as shown in Figure 8c. Let $\mathbf{v}_0$ and $\mathbf{v}_1$ be the two vertices of the edge and $\mathbf{n}_0$ and $\mathbf{n}_1$ be the two neighboring face normals (pointing to the interior of the mesh). $\mathbf{p}$ is in the infeasible region if any of the following is true:

$$(\mathbf{p} - \mathbf{v}_0) \cdot (\mathbf{v}_1 - \mathbf{v}_0) < 0 \tag{2}$$

$$(\mathbf{p} - \mathbf{v}_1) \cdot (\mathbf{v}_0 - \mathbf{v}_1) < 0 \tag{3}$$

$$(\mathbf{p} - \mathbf{s}) \cdot (\mathbf{n}_0 \times (\mathbf{v}_1 - \mathbf{v}_0)) < 0 \tag{4}$$

$$(\mathbf{p} - \mathbf{s}) \cdot (\mathbf{n}_1 \times (\mathbf{v}_0 - \mathbf{v}_1)) < 0 \tag{5}$$

note that $\mathbf{n}_0$ is from the face whose orientation accords to $\mathbf{v}_0 \rightarrow \mathbf{v}_1$.

Face Feasible Region. We can similarly construct the feasible region when $\mathbf{s}$ in on the interior of a face as well. Nonetheless, this particular feasible region test is unnecessary, because when $\mathbf{s}$ is the closest point on the face to $\mathbf{p}$, which is how we pick our candidate boundary points (based on Theorem 2), $\mathbf{p}$ is guaranteed to be in the feasible region.

Our *infeasible region culling* technique performs the tests above and skips the ray traversal if $\mathbf{p}$ is determined to be in the infeasible region, quickly determining that $\mathbf{s}$ cannot be the closest boundary point. Due to numerical precision, the feasible region check can return false results when $\mathbf{p}$ is close to the boundary of the feasible region. There are two types of errors: false positives and false neg-

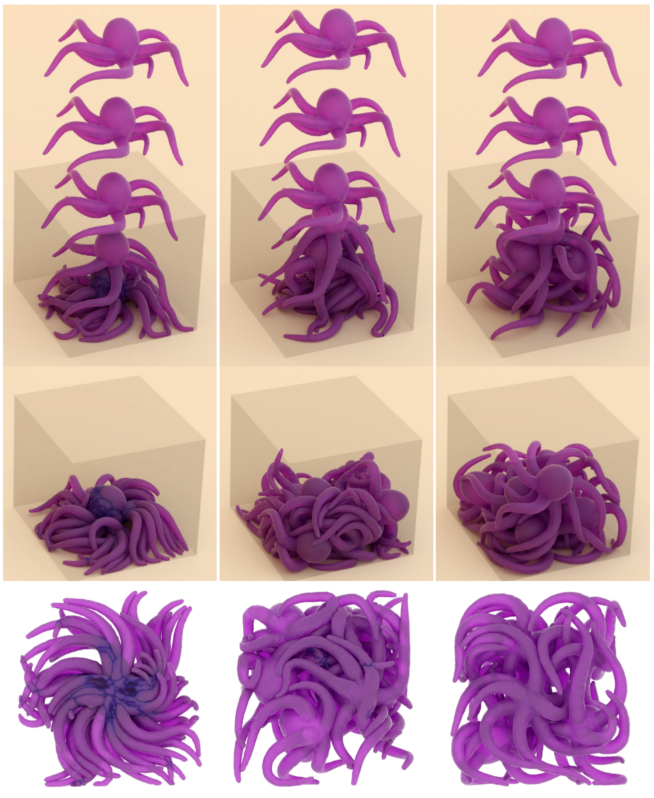


Fig. 9. *Dropping 8 octopi to a box simulated with (a) CCD only, (b) DCD with our shortest path query only, and (c) CCD and DCD with our shortest path query. The bottom row shows the bottom view of the final state. The blue tint highlights the intersecting geometry. The octopus model is from Zhou and Jacobson [2016].*

atives. A false positive is not a big problem: it will only result in an extra traversal. But if a false negative happens, there is a risk of discarding the actual closest surface point. In practice, however, we replace the zeros on the right-hand-sides of the inequalities above with a small negative number ϵ_r to avoid false-positives due to numerical precision limits. In our tests, we have observed that infeasible region culling can provide more than an order of magnitude faster shortest path query.

4 COLLISION HANDLING APPLICATION

As mentioned above, an important application of our method is collision handling with DCD. When DCD finds a penetration, we can use our method to find the closest point on the boundary and apply forces or constraints that would move the penetrating point towards this boundary point.

In our tests with tetrahedral meshes, we use two types of DCD: vertex-tetrahedron and edge-tetrahedron collisions. For vertex-tetrahedron collisions, we find the closest surface point for the colliding vertex. For edge-tetrahedron collisions, we find the center of the part of the edge that intersects with the tetrahedron and then use our method to find the closest surface point to that

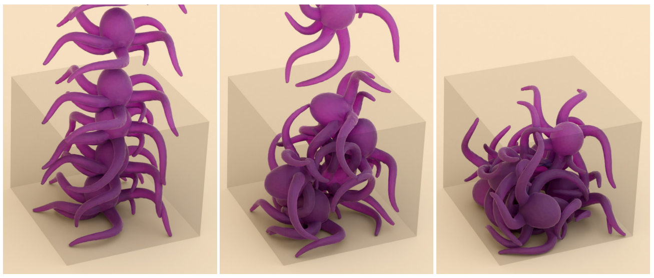


Fig. 10. *Dropping 6 octopi into a box simulated using implicit Euler. This simulation contains 30K vertices and 88K tetrahedra and it takes an average of 15s to simulate each frame. Please see the supplementary material for the details of our implicit Euler framework.*

center point. If an edge intersects with multiple tetrahedra, we choose the intersection center that is closest to the center of the edge. The idea is by keep pushing the center of the edge-tetrahedron intersection towards the surface, which eventually resolves the intersection.

This provides an effective collision handling method with XPBD [Macklin et al. 2016]. Once we find the penetrating point $\mathbf{x}$ we use the standard PBD collision constraint [Müller et al. 2007]

$$c(\mathbf{x}, \mathbf{s}) = (\mathbf{x} - \mathbf{s}) \cdot \mathbf{n} \tag{6}$$

where $\mathbf{s}$ is the closest surface point computed by our method when this collision is from DCD, or the colliding point when it is from CCD, and $\mathbf{n}$ is the surface normal at $\mathbf{s}$. If $\mathbf{s}$ is on a surface edge or vertex, we use the area-weighted average of its neighboring face normals. The XPBD integrator applies projections on each collision constraint c to satisfy $c(\mathbf{x}, \mathbf{s}) \geq 0$. We also apply friction, following Bender et al. [2015]. Please see the supplementary material for the pseudocode of our XPBD framework.

Unlike CCD alone, DCD with our method significantly improves the robustness of collision handling when using a simulation system like XPBD that does not guarantee resolving all collision constraints. This is demonstrated in Figure 9, comparing different collision detection approaches with XPBD. Using only CCD leads to missed collisions when XPBD fails to resolve the collisions detected in previous steps, because CCD can no longer detect them. This quickly results in objects completely penetrating through each other (Figure 9a). Our method with only DCD effectively resolves the majority of collisions (Figure 9b), but it inherits the limitations of DCD. More specifically, using only DCD with sufficiently large time steps and fast enough motion, some collisions can be missed and deep penetrations can resolve the collisions by moving the objects in incorrect directions, again resulting in object parts passing through each other. Furthermore, our method only provides the closest path to the boundary and properly resolving the collisions is left to the simulation system. Unfortunately, XPBD cannot provide any guarantees in collision resolution, so detected penetrations may remain unresolved.

We recommend a hybrid solution that uses both CCD and DCD with our method. This hybrid solution performs DCD in the beginning of the time step to identify the preexisting penetrations or collisions that were not properly resolved in the previous time step.

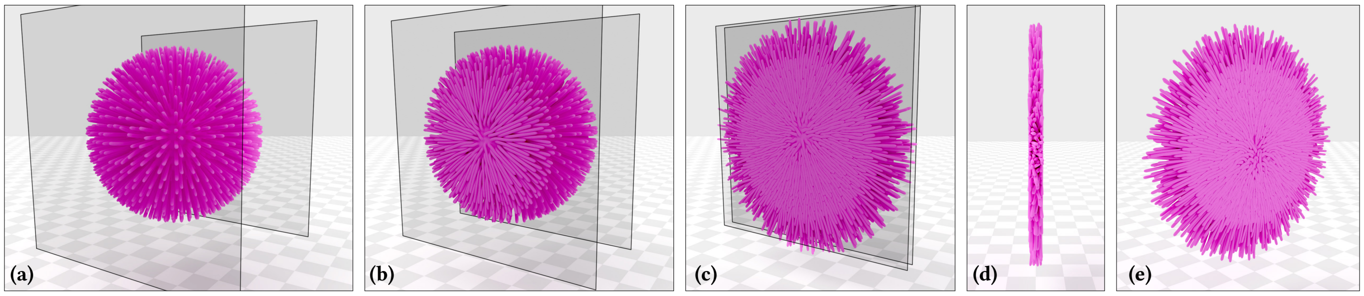


Fig. 11. *Flattening a squishy ball (774K vertices, 2.81M tetrahedra) using two planes. (a-c) the flattening process, (d) Side view of the flattened ball to to* $1/20$ *of its radius, and (e) the other side of the flattened squishy ball.*

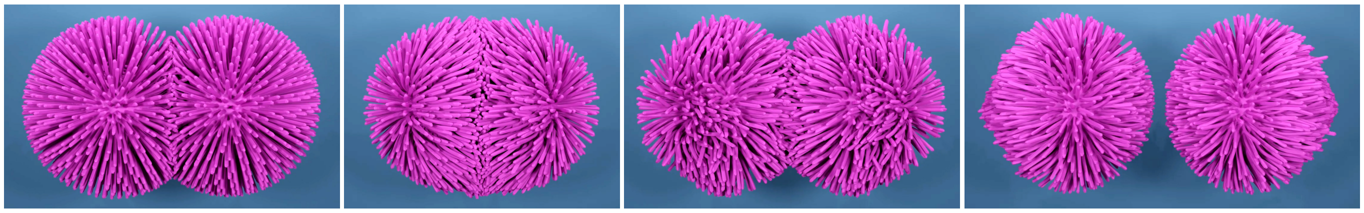


Fig. 12. *Simulation of two squishy balls in head-on collision that come to contact at a relative speed of* $50m/s$. *Both self-collisions and collisions between the two squishy balls are handled using our method.*

The rest of the collisions are detected by CCD without requiring our method to find the closest surface point. The same simulation with this hybrid approach is shown in Figure 9c. Since all penetrations are first detected by CCD and proper collision constraints are applied immediately, deep penetrations become much less likely even with large time steps and fast motion. Yet, this provides no theoretical guarantees. The addition of CCD allows the simulation system to apply collision constraints immediately, before the penetrations become deep, and DCD with our method allows it to continue applying collision constraints when it fails to resolve the initial collision constraints. Note that, while this significantly reduces the likelihood of failed collisions, they can still occur if the simulation system keeps failing to resolve the detected collisions.

The collision handling application of our method is not exclusive to PBD. Our method can also be used with force-based simulation techniques for defining a penalty force with penetration potential energy

$$E_c = \frac{1}{2}\, k \left((\mathbf{p} - \mathbf{s}) \cdot \mathbf{n} \right)^2 \tag{7}$$

where k is the collision stiffness. An example of this is shown in Figure 10.

5 RESULTS

We use XPBD [Macklin et al. 2016] to evaluate our method, because it is one of the fastest simulation methods for deformable objects, providing a good baseline for demonstrating the minor computation overhead introduced by our method. We use mass-spring or NeoHookean [Macklin and Muller 2021] material constraints implemented on the GPU. We handle the collision detection and handling part on the CPU, including the position updates of the collision constraints. We use the hybrid collision detection approach that combines CCD and DCD, as explained above.

We implement both collision detection and closest point query on CPU using Intel's Embree framework [Wald et al. 2014] to create BVH. We generate our timing results on a computer with an AMD Ryzen 5950X CPU, an NVIDIA RTX 3080 Ti GPU, and 64GB of RAM. We acknowledge that our timings are affected by the fact that we copy memory from GPU to CPU every iteration in order to do collision detection and handling, and the whole framework can be further accelerated by implementing the collision detection and shortest path querying on the GPU. As to the parameters of the algorithm, we set ϵ_r to -0.01. ϵ_i is related to the scale and unit of the object, when the object is at a scale of a few meters, we set ϵ_i to 1^{-10}.

5.1 Stress Tests

Figure 11 shows a squishy ball with thin tentacles compressed on two sides and flattened to a thickness that is only $1/20$ of its original radius. Notice that all collisions, including self-intersections of tentacles, are properly resolved even under such extreme compression. Also. the model was able to revert to its original state after the the two planes compressing it were removed.

Figure 12 shows a high-speed head-on collision of two squishy balls. Though the tentacles initially get tangled with frictional-contact right after the collision, all collisions are properly resolved and the two squishy balls bounce back, as expected.

Figure 13 shows two challenging examples of self-collisions caused by twisting a thin beam and two elastic rods. Both instances have shown notable buckling after the twisting. A different frame for the same thin beam is also included in Figure 1. Such self-collisions are particularly challenging for prior self-collision handling methods that pre-split the model into pieces, since it is unclear where the self-collisions might occur before the simulation.

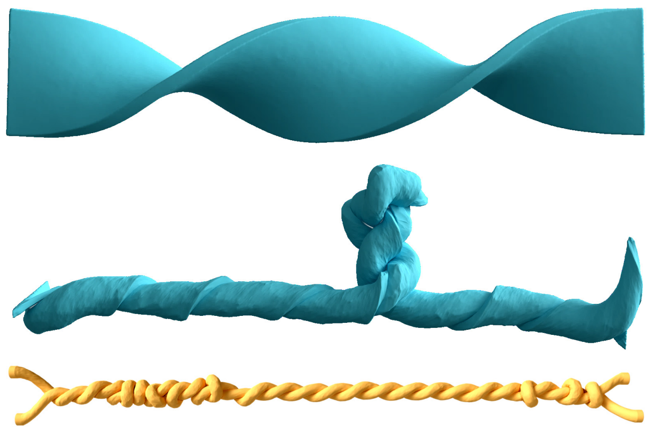


Fig. 13. *Twisting (top-middle) a thin beam with 400K vertices & 1.9M tetrahedra, and (bottom) two elastic rods with 281K vertices & 1.3M tetrahedra, demonstrating unpredictable self-collisions and buckling.*

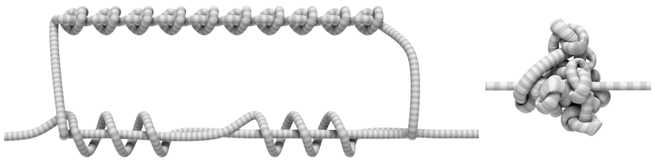
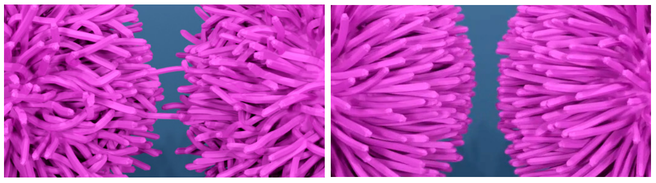
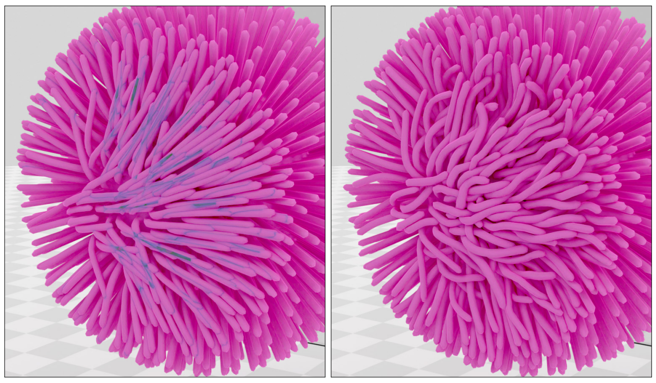


Fig. 14. *Simulation of a nested knot starting from (left) the initial state to (right) the final knot.*



(a) *Naive collisions between objects* **(b)** *Our inter-object collisions*

Fig. 15. *Two squishy balls in head-on collision comparing collision handling between two different objects (a) by naively accepting the Euclidean closest point as the closest boundary point and (b) with our method. All self-collisions are handled using our method in both cases.*



(a) *Self-collisions off* **(b)** *Self-collisions turned on*

Fig. 16. *Simulation of a squishy ball compressed from either side, as in Figure 11, (a) with self-collisions turned off to produce a state with many complex self-collisions, where the blue tint highlights deeper penetrations caused by subsurface scattering artifacts due to intersecting geometry, and (b) a few frames after self-collisions are turned on, showing that our method with XPBD quickly recovers them.*

Another challenging self-collision case is shown in Figure 14, where nested knots were formed by pulling an elastic rod from both sides. In this case, there is a significant amount of sliding frictional self-contact, changing pairs of elements that collide with each other. Though a substantial amount of force is applied near the end, the simulation is able to form a stable knot.

Figure 15a shows the same experiment using naive closest boundary point computation for the collisions between the two squishy balls (by picking the closest boundary point on the other object purely based on Euclidean distances), only handling self-collisions with our method. Notice that it includes (temporarily) entangled tentacles between the two squishy balls and visibly more deformations of tentacles elsewhere, as compared to using our method (Figure 15b). This is because, in the presence of self-collisions, naively handling closest boundary point queries between different objects is prone to picking incorrect boundary points that do not resolve the collision, resulting in prolonged contact and inter-locking.

5.2 Solving Existing Intersections

Our method can successfully resolve existing self collisions. A demonstration of this is provided in Figure 17. In this example, the initial state (Figure 17a) is generated by dropping a noodle model without handling self-collisions. When we turn on self-collisions, all existing self-intersections are quickly resolved within 10 substeps (Figure 17b), resulting in numerous inverted elements due to strong collision constraints. Then, the simulation resolves them (Figure 17c) and finally the model comes to a rest with self-contact (Figure 17d). In this experiment, we perform collision projections for all vertices (not only for surface vertices) and all tetrahedra's centroids to resolve the intersection in the completely overlapping parts. Note that we do not provide a theoretical guarantee to resolve all the existing intersections. In practice, however, in all our tests all collisions are resolved after just a few iterations/substeps.

Another example is shown in Figure 16, generated by compressing a squishy ball with two planes on either side, similar to Figure 11 but without handling self-collisions. This results in a significant number of complex unresolved self-collisions (Figure 16a), which are quickly resolved within a few substeps when self-collision handling is turned on (Figure 16b).

5.3 Large-Scale Experiments

An important advantage of our method is that, by providing a robust collision handling solution, we can use fast simulation techniques for scenarios involving a large number of objects and complex collisions. An example of this is demonstrated in Figure 18, showing 600 deformable octopus models forming a pile. Due to its complex geometry, the octopus model can cause numerous self-collisions

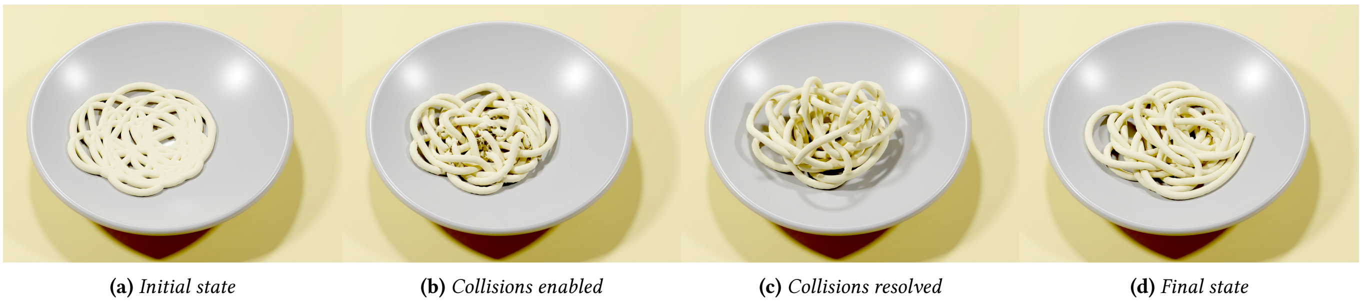


Fig. 17. *Solving existing collisions. Starting from (a) an initial state with many self-collisions, (b) after collision handling is enabled, (c) our method can quickly resolve them, and (d) achieve a self-collision-free final state.*

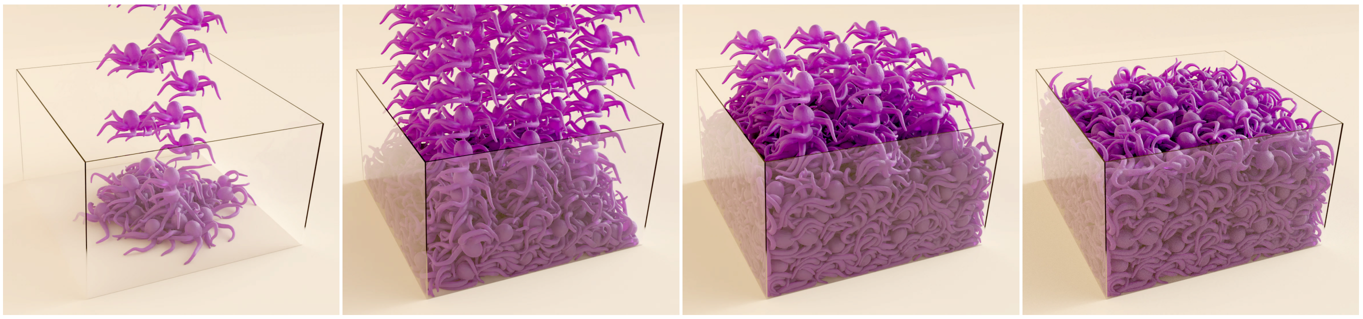


Fig. 18. *600 deformable octopus models (3.1M vertices and 8.88M tetrahedra in total) dropped into a container, forming a pile with collisions.*

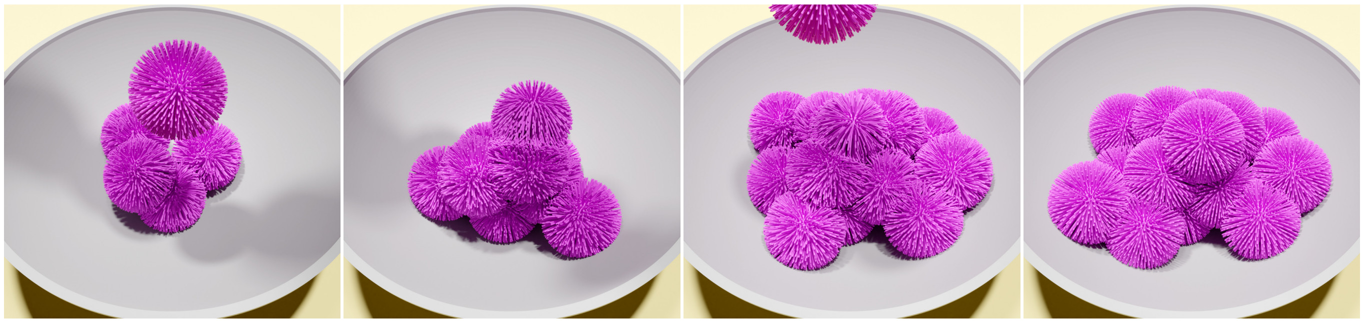


Fig. 19. *Simulation of 16 squishy balls (a total of 11.2 million tetrahedra) dropped into a bowl, forming a stable pile with active collisions.*

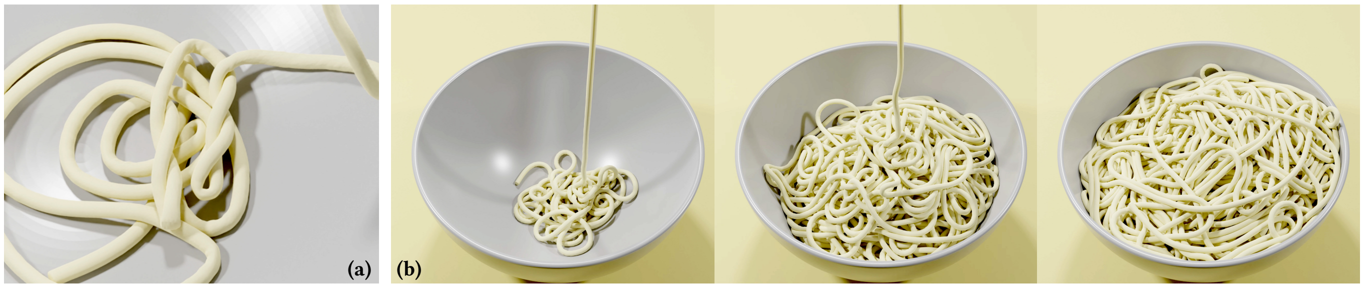


Fig. 20. *Simulation of a long noodle (a) presenting unpredictable complex self-collisions and (b) forming a large pile with self-collisions.*

and inter-object collisions. Both collision types are handled using our method. At the end of the simulation, a stable pile is formed with 185K active collisions per time step.

Figure 19 shows another large-scale experiment involving 16 squishy balls. Another frame from this simulation is also included in Figure 1. At the end of the simulation, the squishy balls form a stable

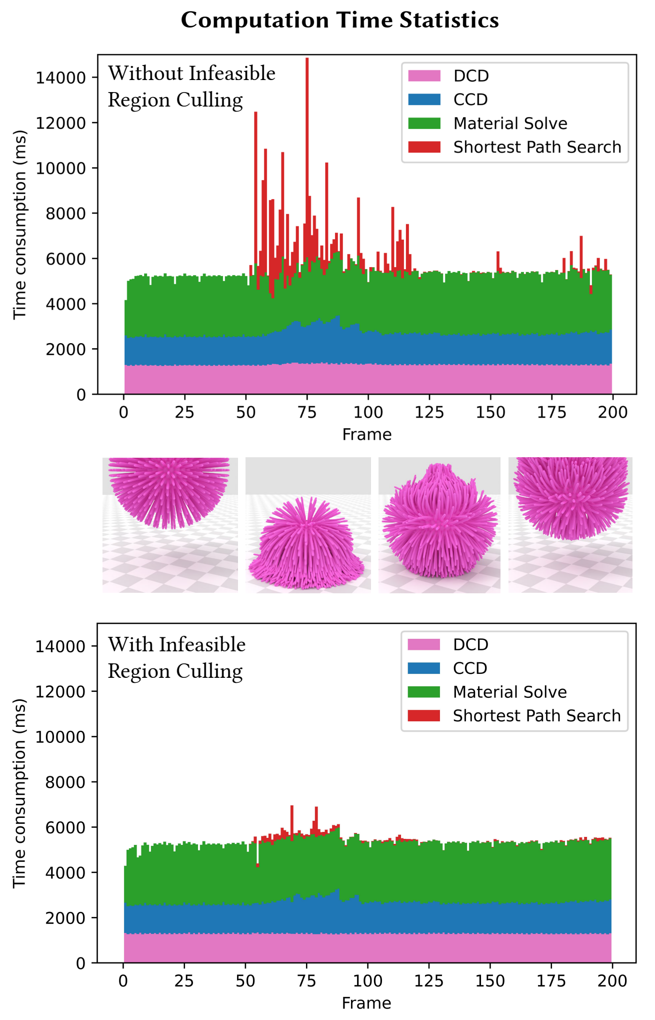
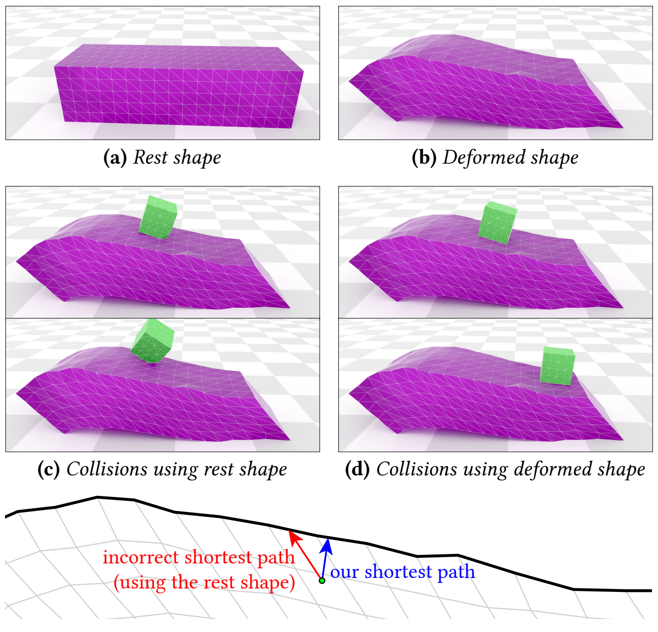


Fig. 21. *The computation time statistics of each simulation component on stacked bar charts: (top) without infeasible region culling and (bottom) with infeasible region culling. The middle row shows the simulation at frames 0, 75, 125, 175 respectively.*



(a) *Rest shape* **(b)** *Deformed shape*

(c) *Collisions using rest shape* **(d)** *Collisions using deformed shape*

(e) *The incorrect shortest path using the rest shape vs. ours*

Fig. 22. *Comparison between the rest pose closest boundary point and our closest boundary point. (a) The rest pose the cuboid model. (b) We deform the cuboid to a certain shape, then drop a cube on top of it. (c) In the simulation using the rest pose closest boundary point, the cube got incorrectly pulled up. (d) Using our exact closest point, the cube successfully slides down. (e) The shortest path to the surface for an example point, showing that using the closest surface point queried from the rest shape results in an incorrect and longer path.*

pile and remain in rest-in-contact with active self-collisions (12K) and inter-object collisions (125K) between neighboring squishy balls.

We also include an experiment with a single long noodle piece in Figure 20 that is dropped into a bowl. This simulation forms numerous complex and unpredictable self-collisions (Figure 20a). At the end of the simulation, we achieve a stable pile with 104K active self-collisions per time step in this example. Figure 1 includes a rendering of this final pile without the bowl and a cross-section view, showing that the interior self-collisions are properly resolved.

5.4 Performance

We provide the performance numbers for the experiments above in Table 1. Notice that, even though we are using a highly efficient material solver that is parallelized on the GPU, our method provides a relatively small overhead. This includes some highly-challenging experiments, involving a large number of complex collisions. The highest overhead of our method is in experiments in which deliberately disabled self-collisions to form a large number of complex self-collisions. Note that all collision detection and handling computations are performed on the CPU, and a GPU implementation would likely result in a smaller overhead.

We demonstrate the effect of our infeasible region culling by simulating a squishy ball dropped to the ground with and without this acceleration. The computation time breakdown of all frames are visualized in Figure 21. In this example, using our infeasible region culling, the shortest path query gains a speed-up of 10-30× for some frames, providing identical results. Additionally, the accelerated shortest path query results in a more uniform computation time, avoiding the peaks visible in the graph.

5.5 Comparisons to Rest Shape Shortest Paths

A popular approach in prior work for handling self-collisions is using the rest shape of the model that does not contain self-collisions for performing the shortest path queries. This makes the computation much simpler, but obviously results in incorrect shortest boundary paths. With sufficient deformations, these incorrect boundary paths can lead to large enough errors and instabilities.

Table 1. *Performance results. Time step size and frame times are given in seconds, where frame times are measured at 60 FPS. Operations Q., Tr., and Tet. represent the number of BVH queries, traversals, and total tetrahedra visited on average per time step, respectively.*

		Number of		Avrg. Collisions		Avrg. Operations			Time Step	Frame Time		Average Time %			
		Vert.	Tet.	CCD	DCD	Q.	Tr.	Tet.	Size × Iter.	Avrg.	Max.	XPBD	CCD	DCD	**Ours**
Flattened Squishy Ball	(Fig. 11)	774 K	2.81 M	16.8 K	7.1 K	56	6.6	5.2	3.3e-4 × 3	10.89	18.04	30.9 %	29.0 %	31.7 %	**8.3 %**
Twisted Thin Beam	(Fig. 13)	400 K	1.9 M	8.3 K	3.1 K	45	5.7	7.2	3.3e-4 × 3	8.16	15.92	29.7 %	31.3 %	32.1 %	**6.9 %**
Twisted Rods	(Fig. 13)	281 K	1.3 M	4.8 K	2.6 K	33	5.3	4.7	3.3e-4 × 3	5.25	11.17	42.1 %	26.9 %	27.0 %	**4.0 %**
Nested Knots	(Fig. 14)	38.1 K	103 K	3.1 K	0.6 K	31	4.2	4.4	5.5e-4 × 3	0.25	0.32	61.8 %	23.3 %	9.5 %	**5.2 %**
2 Squishy Balls	(Fig. 12)	418 K	1.4 M	22.4 K	1.3 K	36	9.6	11.3	3.3e-4 × 3	1.96	2.87	52.2 %	28.4 %	18.5 %	**10.9 %**
Pre-Intersect. Noodle	(Fig. 17)	40 K	110 K	N/A	15.2 K	65	12.0	13.6	8.3e-4 × 3	0.21	0.45	51.6 %	17.6 %	18.4 %	**12.4 %**
Pre-Intersect. Squishy Ball	(Fig. 16)	219 K	704 K	N/A	45.8 K	89	12.0	14.0	3.3e-4 × 3	1.54	2.63	44.3 %	18.2 %	19.1 %	**18.4 %**
600 Octopi	(Fig. 18)	3.1 M	8.88 M	104.0 K	6.4 K	12	3.6	4.1	8.3e-4 × 3	16.40	17.90	68.3 %	15.4 %	13.4 %	**2.9 %**
16 Squishy Balls	(Fig. 19)	3.5 M	11.2 M	118.5 K	8.5 K	29	4.5	6.6	3.3e-4 × 3	18.50	20.20	49.3 %	25.0 %	21.8 %	**3.9 %**
Long Noodle	(Fig. 20)	860 K	2.29 M	102.6 K	6.1 K	11	3.6	3.2	8.3e-4 × 3	4.10	4.50	67.6 %	14.8 %	14.9 %	**2.7 %**
8 Octopi CCD Only	(Fig. 9a)	40 K	118 K	2.1 K	N/A	N/A	N/A	N/A	3.3e-3 × 5	0.028	0.036	86.6 %	13.4 %	N/A	N/A
8 Octopi DCD Only	(Fig. 9b)	40 K	118 K	N/A	2.4 K	13	3.7	4.1	3.3e-3 × 5	0.038	0.045	79.2 %	N/A	10.7 %	**10.1 %**
8 Octopi hybrid	(Fig. 9c)	40 K	118 K	2.3 K	0.2 K	11	3.3	3.9	3.3e-3 × 5	0.035	0.038	79.3 %	10.1 %	9.6 %	**1.0 %**

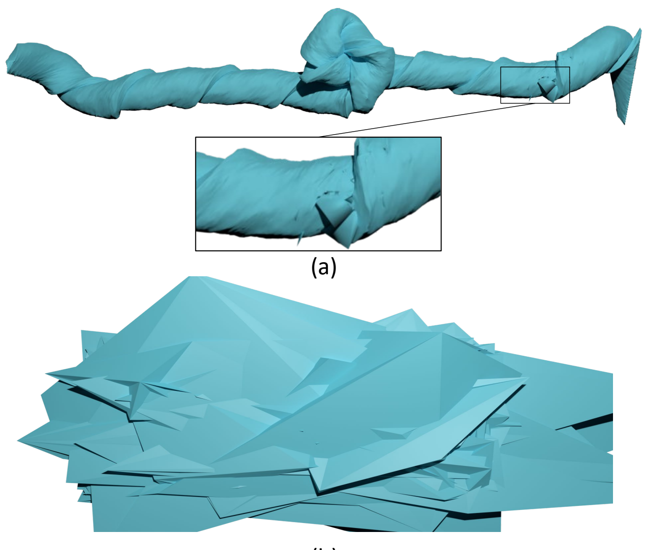


Fig. 23. *Simulation of twisting a thin beam, shown in Figure 13, soon after replacing our method with using the rest pose for finding the closest boundary point: (a) instabilities caused by incorrect closest boundary points found using this approach, and (b) exploded simulation after a few frames.*

Figure 22 shows a simple example, where a small cube is dropped onto a deformed object. Notice that the rest shape of the object (Figure 22a) is sufficiently different from the deformed shape (Figure 22b). With collision handling using this rest shape, the cube moves against gravity and eventually bounces back (Figure 22c), instead of sliding down the surface, as simulated using our method (Figure 22d). Figure 22e shows a 2D illustration of example shortest paths generated by both methods. Notice that using the rest shape results in a longer path to the surface that corresponds to higher collision energy. In contrast, our method minimizes the collision energy by using the actual shortest path to the boundary.

Figure 23 shows a more complex example with self-collisions that is initially simulated using our method (Figure 13) until complex self-collisions are formed. When we switch to using the rest shape to find the boundary paths, the simulation explodes following a number of incorrectly-handled self-collisions.

In general, using the rest shape not only generates incorrect shortest boundary paths, but also injects energy into the simulation. This is because an incorrect shortest boundary path is, by definition, longer than the actual shortest boundary path, thereby corresponds to higher potential energy.

6 DISCUSSION

An important advantage of our method is that it can work with simulation systems that do not provide any guarantees about resolving collisions. Therefore, we can use fast simulation techniques like XPBD to handle complex scenarios involving numerous self-collisions, as demonstrated above.

Yet, our method cannot handle all types of self-collisions and it requires a volumetric mesh. We cannot handle collisions of codimensional objects, such as cloth or strands. Our method would also have difficulties handling meshes with thin volumes or no interior elements.

Our method is essentially a shortest boundary path computation method. It is based on the fact that an interior point's shortest path to the boundary is always a line segment. This assumption always holds for objects like tetrahedral meshes in 3D or triangular mesh in 2D Euclidean space. Therefore, our method cannot handle shortest boundary paths in non-Euclidean spaces, such as geodesic paths on surfaces in 3D.

Using our method for collision handling with DCD inherits the limitations of DCD. For example, when with large time steps and sufficiently fast motion, penetration can get too deep, and the shortest boundary path may be on the other side of the penetrated model, causing undesirable collision handling. In practice, this problem can be efficiently solved by coupling CCD and DCD, as we demonstrate with our results above.

7 CONCLUSION

We have presented a formal definition of the shortest path to boundary in the context of self-intersections and introduced an efficient and robust algorithm for finding the exact shortest boundary paths for meshes. We have shown that this approach provides an effective solution for handling both self-collisions and inter-object collisions using DCD in combination with CCD, using a simulation system that does not provide any guarantees about resolving the collision constraints. Our results show highly complex simulation scenarios involving collisions and rest-in-contact conditions that are properly handled with our method with a relatively small computational overhead.

ACKNOWLEDGMENTS

We thank Alper Sahistan, Yin Yang, and Kui Wu for their helpful comments and suggestions. We also thank Alec Jacobson and Tiantian Liu for providing online volumetric mesh datasets. This project was supported in part by NSF grant #1764071.

REFERENCES

Jérémie Allard, François Faure, Hadrien Courtecuisse, Florent Falipou, Christian Duriez, and Paul G Kry. 2010. Volume contact constraints at arbitrary resolution. In *ACM SIGGRAPH 2010 papers*. 1–10.

Aytek Aman, Serkan Demirci, and Uğur Güdükbay. 2022. Compact tetrahedralization-based acceleration structures for ray tracing. *Journal of Visualization* (2022), 1–13.

Mukund Balasubramanian, Jonathan R Polimeni, and Eric L Schwartz. 2008. Exact geodesics and shortest paths on polyhedral surfaces. *IEEE transactions on pattern analysis and machine intelligence* 31, 6 (2008), 1006–1016.

David Baraff. 1994. Fast contact force computation for nonpenetrating rigid bodies. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. 23–34.

Ted Belytschko and Mark O Neal. 1991. Contact-impact by the pinball algorithm with penalty and Lagrangian methods. *Internat. J. Numer. Methods Engrg.* 31, 3 (1991), 547–572.

Jan Bender, Matthias Müller, and Miles Macklin. 2015. Position-Based Simulation Methods in Computer Graphics.. In *Eurographics (tutorials)*. 8.

Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2014. Projective dynamics: Fusing constraint projections for fast simulation. *ACM transactions on graphics (TOG)* 33, 4 (2014), 1–11.

Stephen Cameron. 1997. Enhancing GJK: Computing minimum and penetration distances between convex polyhedra. In *Proceedings of international conference on robotics and automation*, Vol. 4. IEEE, 3112–3117.

John Canny. 1986. Collision detection for moving polyhedra. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 2 (1986), 200–209.

Jindong Chen and Yijie Han. 1990. Shortest paths on a polyhedron. In *Proceedings of the sixth annual symposium on Computational geometry*. 360–369.

Keenan Crane, Marco Livesu, Enrico Puppo, and Yipeng Qin. 2020. A Survey of Algorithms for Geodesic Paths and Distances. *arXiv preprint arXiv:2007.10430* (2020).

Ounan Ding and Craig Schroeder. 2019. Penalty force for coupling materials with Coulomb friction. *IEEE transactions on visualization and computer graphics* 26, 7 (2019), 2443–2455.

Evan Drumwright. 2007. A fast and stable penalty method for rigid body simulation. *IEEE transactions on visualization and computer graphics* 14, 1 (2007), 231–240.

Zachary Ferguson, Minchen Li, Teseo Schneider, Francisca Gil Ureta, Timothy R Langlois, Chenfanfu Jiang, Denis Zorin, Danny M Kaufman, and Daniele Panozzo. 2021. Intersection-free rigid body dynamics. *ACM Trans. Graph.* 40, 4 (2021), 183–1.

Susan Fisher and Ming C Lin. 2001a. Deformed distance fields for simulation of non-penetrating flexible bodies. In *Computer Animation and Simulation 2001*. Springer, 99–111.

Susan Fisher and Ming C Lin. 2001b. Fast penetration depth estimation for elastic bodies using deformed distance fields. In *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No. 01CH37180)*, Vol. 1. IEEE, 330–336.

Marie-Paule Gascuel. 1993. An implicit formulation for precise contact modeling between flexible solids. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 313–320.

James K Hahn. 1988. Realistic animation of rigid bodies. *ACM Siggraph computer graphics* 22, 4 (1988), 299–308.

Bruno Heidelberger, Matthias Teschner, Richard Keiser, Matthias Müller, and Markus H Gross. 2004. Consistent penetration depth estimation for deformable collision response.. In *VMV*, Vol. 4. 339–346.

Everton Hermann, François Faure, and Bruno Raffin. 2008. Ray-traced collision detection for deformable bodies. In *GRAPP 2008-3rd International Conference on Computer Graphics Theory and Applications*. INSTICC, 293–299.

Gentaro Hirota, Susan Fisher, and Ming Lin. 2000. Simulation of non-penetrating elastic bodies using distance fields. *University of North Carolina at Chapel Hill Technical Report: TR00-018. Spring* (2000).

I Huněk. 1993. On a penalty formulation for contact-impact problems. *Computers & structures* 48, 2 (1993), 193–203.

Changsoo Je, Min Tang, Youngeun Lee, Minkyoung Lee, and Young J Kim. 2012. PolyDepth: Real-time penetration depth computation using iterative contact-space projection. *ACM Transactions on Graphics (TOG)* 31, 1 (2012), 1–14.

Ladislav Kavan. 2003. Rigid body collision response. *Vectors* 1000, 2 (2003).

Dan Koschier, Crispin Deul, Magnus Brand, and Jan Bender. 2017. An hp-adaptive discretization algorithm for signed distance field generation. *IEEE transactions on visualization and computer graphics* 23, 10 (2017), 2208–2221.

Ares Lagae and Philip Dutré. 2008. Accelerating ray tracing using constrained tetrahedralizations. In *Computer Graphics Forum*, Vol. 27. Wiley Online Library, 1303–1312.

Lei Lan, Danny M. Kaufman, Minchen Li, Chenfanfu Jiang, and Yin Yang. 2022a. Affine Body Dynamics: Fast, Stable and Intersection-Free Simulation of Stiff Materials. *ACM Trans. Graph.* 41, 4, Article 67 (jul 2022), 14 pages. https://doi.org/10.1145/3528223.3530064

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

Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M Kaufman. 2020. Incremental potential contact: Intersection-and inversion-free, large-deformation dynamics. *ACM transactions on graphics* (2020).

Yijing Li and Jernej Barbič. 2018. Immersion of self-intersecting solids and surfaces. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–14.

Yong-Jin Liu. 2013. Exact geodesic metric in 2-manifold triangle meshes using edge-based data structures. *Computer-Aided Design* 45, 3 (2013), 695–704.

Miles Macklin, Kenny Erleben, Matthias Müller, Nuttapong Chentanez, Stefan Jeschke, and Zach Corse. 2020. Local optimization for robust signed distance field collision. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3, 1 (2020), 1–17.

Miles Macklin and Matthias Muller. 2021. A Constraint-based Formulation of Stable Neo-Hookean Materials. In *Motion, Interaction and Games*. 1–7.

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*. 49–54.

Maxime Maria, Sébastien Horna, and Lilian Aveneau. 2017. Efficient ray traversal of constrained Delaunay tetrahedralization. In *12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2017)*, Vol. 1. 236–243.

Gerd Marmitt and Philipp Slusallek. 2006. Fast ray traversal of tetrahedral and hexahedral meshes for direct volume rendering. In *Proceedings of the Eighth Joint Eurographics/IEEE VGTC conference on Visualization*. 235–242.

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. In *ACM SIGGRAPH 2011 papers*. 1–12.

Brian Mirtich and John Canny. 1995. Impulse-based simulation of rigid bodies. In *Proceedings of the 1995 symposium on Interactive 3D graphics*. 181–ff.

Joseph SB Mitchell, David M Mount, and Christos H Papadimitriou. 1987. The discrete geodesic problem. *SIAM J. Comput.* 16, 4 (1987), 647–668.

Nathan Mitchell, Mridul Aanjaneya, Rajsekhar Setaluri, and Eftychios Sifakis. 2015. Non-manifold level sets: A multivalued implicit surface representation with applications to self-collision processing. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 1–9.

Matthew Moore and Jane Wilhelms. 1988. Collision detection and response for computer animation. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*. 289–298.

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.

Carol O'Sullivan and John Dingliana. 1999. Real-time collision detection and response using sphere-trees. (1999).

Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Charles Hansen, and Peter Shirley. 2005. Interactive ray tracing for volume visualization. In *ACM SIGGRAPH 2005 Courses*. 15–es.

John C Platt and Alan H Barr. 1988. Constraints methods for flexible models. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*. 279–288.

Stéphane Redon and Ming C. Lin. 2006. A Fast Method for Local Penetration Depth Computation. *Journal of Graphics Tools* 11 (2006), 37 – 50.

Alper Şahıstan, Serkan Demirci, Nathan Morrical, Stefan Zellmann, Aytek Aman, Ingo Wald, and Uğur Güdükbay. 2021. Ray-traced shell traversal of tetrahedral meshes for direct volume visualization. In *2021 IEEE Visualization Conference (VIS)*. IEEE, 91–95.

Vitaly Surazhsky, Tatiana Surazhsky, Danil Kirsanov, Steven J Gortler, and Hugues Hoppe. 2005. Fast exact and approximate geodesics on meshes. *ACM transactions on graphics (TOG)* 24, 3 (2005), 553–560.

Yun Teng, Miguel A Otaduy, and Theodore Kim. 2014. Simulating articulated subspace self-contact. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 1–9.

Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. 1987. Elastically deformable models. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques.* 205–214.

Mickeal Verschoor and Andrei C Jalba. 2019. Efficient and accurate collision response for elastically deformable models. *ACM Transactions on Graphics (TOG)* 38, 2 (2019), 1–20.

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

Bin Wang, François Faure, and Dinesh K Pai. 2012. Adaptive image-based intersection volume. *ACM Transactions on Graphics (TOG)* 31, 4 (2012), 1–9.

Bolun Wang, Zachary Ferguson, Teseo Schneider, Xin Jiang, Marco Attene, and Daniele Panozzo. 2021. A Large-scale Benchmark and an Inclusion-based Algorithm for Continuous Collision Detection. *ACM Transactions on Graphics (TOG)* 40, 5 (2021), 1–16.

Shi-Qing Xin and Guo-Jin Wang. 2009. Improving Chen and Han's algorithm on the discrete geodesic problem. *ACM Transactions on Graphics (TOG)* 28, 4 (2009), 1–8.

Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016).