Shortest Path to Boundary for Self-Intersecting Meshes

Supplemental Document

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

This supplemental document includes the pseudocode of our algorithm and proof of theorems.

1 IMPLEMENTATION

1.1 Algorithms

We show the pseudocode of our algorithm to determine whether a line segment is a valid path in Algorithm 1, the ray-triangle intersection algorithm in Algorithm 2, the shortest path query algorithm in line 18 and the infeasible region culling algorithm in Algorithm 4. We provide the proof of theorems in Section 2.

1.1.1 Explanation of the Tetrahedral Traverse Algorithm. Here we discuss some details of Algorithm 1. We import some techniques from the field of tetrahedral traverse based volumetric rendering to accelerate the tetrahedral traverse procedure [Aman et al. 2022], in which they construct a 2D coordinate system for each ray [Duff et al. 2017] and determine the ray-triangle intersection based on it. This drastically reduced the number of arithmetic operations. Nevertheless, there are some robustness issues associated with tetrahedral traverse that are still unsolved, such as dead ends and infinite loops. In [Aman et al. 2022], they just discard the ray if it forms a loop because one ray does not matter much among billions of rays running in parallel. In our case though, we can not do that because that very ray may lead to the actual global geodesic path we are looking for. Instead, we try to recover from an earlier state and get out of the loop the other way. In addition, since we need to handle the case of the inverted tetrahedron and the case of the ray going backward, we modified their 2D ray-triangle intersecting algorithm to take the orientation of the incoming into face consideration, see Algorithm 2.

We found that the tetrahedral traverse only forms a loop when the ray is hitting near a vertex or edge of the tetrahedron. As illustrated by Figure 1, the ray is trying to get out of tetrahedron t_0. However, the ray is intersecting with a vertex of t_0, and the ray-triangle intersection algorithm is determining the ray intersecting with f_0. Then the algorithm will go to tetrahedron t_1, and similarly it goes to t_2 and t_3 through f_1 and f_2. However, when at t_3, the ray-triangle intersection algorithm can determine the ray intersects with f_3 and put the algorithm back to t_0 and thus forms an infinite loop: $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_0 \rightarrow \ldots$, and the algorithm will be stuck at this vertex. To determine whether the traverse has formed a loop, we maintain a set $\mathcal{T}$ that records all the traversed tetrahedra, as shown in line 1 of Algorithm 1, and do loop check every iteration.

After the loop is detected, we need to be able to recover from an earlier state where the loop has not been formed. This is achieved by managing a candidate intersecting face stack F, see Algorithm 1. As shown in Algorithm 2, we use a constant positive parameter ϵ_i to relax the ray-triangle intersection. We regard a ray as intersecting with a face if its close enough to its boundary. Thus a ray can intersect with multiple faces of a tet. For each tetrahedron we traverse through, we find all its intersecting faces except the incoming face and put them to the stack F, as shown in line 28~32 of Algorithm 1.

ALGORITHM 1: TetrahedralTraverse

```
Input: s: A surface point of $M$, $t_0$: the tetrahedron that s belongs to,
       p: the target internal point, $f_0$: surface triangle containing s,
       the deformed model $M$.
Output: Boolean value representing whether there is a valid
        tetrahedral traverse between s and p.
1  $\mathcal{T} = \emptyset$ ;
2  $\mathbf{r} = \mathbf{b} - \mathbf{a}$ ;
3  $\mathbf{u}, \mathbf{v}$ =generateRayCoordinateSystem($\mathbf{r}$) [Duff et al. 2017];
4  $F$ =exitFaceSelection($t_0$, $f_0$, $\mathbf{u}$, $\mathbf{v}$, $\mathbf{s}$);;
5  $T = \emptyset$ ;
6  for $f$ in $F$ do
7      $t$ =the tetrahedron on the other side of $f$;
8      $T$.push_back($t$);
9  end
10 while $F \neq \emptyset$ do
11     $f = F$.back();
12     $F$.pop_back();
13     $t = T$.back();
14     $T$.pop_back();
15     if $t \in \mathcal{T}$ then
16         continue;
17     else
18         $\mathcal{T} = \mathcal{T} \cup \{t\}$;
19     if $\mathbf{p} \in t$ then
20         return True;
21     if $f$ is a surface triangle then
22         return False;
23     $\mathbf{c}$ =the intersecting point on $f$;
24     if intersection free then
25         if $|\mathbf{c} - \mathbf{s}| > |\mathbf{p} - \mathbf{s}|$ then
26             return False;
27     $F_{\text{new}}$ = ExitFaceSelection($t$, $f$, $\mathbf{u}$, $\mathbf{v}$, $\mathbf{s}$);
28     for $f'$ in $F_{new}$ do
29         $t$ =the tetrahedron on the other side of $f'$;
30         $T$.push_back($t$);
31         $F$.push_back($f'$);
32     end
33     return False;
34 end
```

SIGGRAPH, August 6, 2023, Los Angeles, CA, USA
2023. ACM ISBN 978-1-4503-1234-5/22/07...$15.00
https://doi.org/10.1145/8888888.7777777

ALGORITHM 2: ExitFaceSelection

```
Input: $t$: the current tetrahedron; $f$: the incoming face of the ray; $\mathbf{u}$, $\mathbf{v}$: coordinate basis; $\mathbf{s}$: ray origin.
Output: possible exiting faces of the ray.
$\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ = 3 vertices on $f$, ordered according to $f$'s orientation;
$\mathbf{p}_3$ = the vertex of $t$ that is not on $f$;
$\mathbf{p}'_0, \mathbf{p}'_1, \mathbf{p}'_2, \mathbf{p}'_3$ = projectTo2D($\mathbf{p}_0 \sim \mathbf{p}_3$, $\mathbf{u}$, $\mathbf{v}$, $\mathbf{s}$);
sign = signOf(($\mathbf{p}'_1 - \mathbf{p}'_0) \times (\mathbf{p}'_2 - \mathbf{p}'_0)$);
$F_{\text{exit}} = \emptyset$ ;
if $sign \cdot det(\mathbf{p}'_3, \mathbf{p}'_1) \geq -\epsilon_i$ and $sign \cdot det(\mathbf{p}'_3, \mathbf{p}'_2) \leq \epsilon_i$ then
    $F_{\text{exit}} = F_{\text{exit}} \cup \{f_0\}$
if $sign \cdot det(\mathbf{p}'_3, \mathbf{p}'_2) \geq -\epsilon_i$ and $sign \cdot det(\mathbf{p}'_3, \mathbf{p}'_0) \leq \epsilon_i$ then
    $F_{\text{exit}} = F_{\text{exit}} \cup \{f_1\}$
if $sign \cdot det(\mathbf{p}'_3, \mathbf{p}'_0) \geq -\epsilon_i$ and $sign \cdot det(\mathbf{p}'_3, \mathbf{p}'_1) \leq \epsilon_i$ then
    $F_{\text{exit}} = F_{\text{exit}} \cup \{f_2\}$
return $F_{\text{exit}}$
```

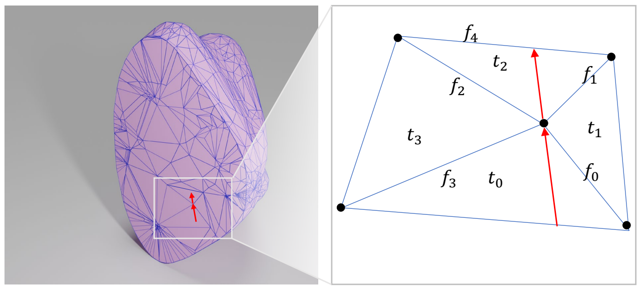

Fig. 1. *(a) A clip view of a tetrahedral mesh, where a ray (marked by red arrows) is passing exactly through a vertex of a tetrahedron. (b) Zoom in to the section where the ray-vertex intersection happens.*

We also manage another stack T to store the tetrahedron on the other side of the face, which always has the same size as F. When the ray intersects with one of the vertices of the tetrahedron, we can end up putting more than one face to F. Then we have a while loop that pops a face at each iteration from the end of F and goes to the tetrahedron on the other side of f by popping the tetrahedron at the end of T, and we repeat this procedure to add more faces and tetrahedra to the stack. Since each time the algorithm will only pop the newest element from the stack, the traverse will perform a depth-first-search-like behavior when no loop is detected.

In the case of a loop is detected, the algorithm will pick an intersecting face from F and a tetrahedron from T which are added from a previously visited tetrahedron and continue going, see lines 11~14 of Algorithm 1. Since the ray is not allowed to go back to a tetrahedron that it has visited, this guarantees the algorithm will not fall into an infinite loop.

1.1.2 Finding the Shortest Path to the Surface. We propose an efficient algorithm that searches for the shortest path to the surface for an interior point $\mathbf{p}$ inside a tetrahedral mesh, see line 18. With a spatial partition algorithm, we can partition all the surface elements (i.e., all the surface triangles) of M using a spatial partition structure (e.g., bounding box hierarchy). Then we can start a point query

ALGORITHM 3: ShortestPathToSurface

```
Input: An interior point $\mathbf{p}$, a penetrated surface point $\mathbf{s}$ that is overlapping with $\mathbf{p}$.
Output: $\mathbf{p}$'s shortest path to the surface.
$r = inf$;
$\mathcal{S}$ = {all the surface triangles$f$};
while $S \neq \emptyset$ do
    $f$ = do primitives query for $\mathcal{S}$ centered at $\mathbf{p}$ with radius $r$;
    $\mathbf{s}'$ = $\mathbf{p}$'s Euclidean closest point on $f$;
    if $\mathbf{s} = \mathbf{s}'$ then
        continue;
    if $TetrahedralTraverse(\mathbf{s}', \mathbf{p}, f)$ then
        $r = \|\mathbf{s}' - \mathbf{p}\|$;
        $\mathcal{S} = \mathcal{S} \setminus \{f\}$ ;
        $\mathbf{s}_{\text{closest}} = \mathbf{s}$ ;
end
return $l(\mathbf{s}_{\text{closest}}, \mathbf{p})$
```

centered at $\mathbf{p}$ using an infinite query radius, see line 1 and line 4 of line 18, which will return all the surface elements sequentially in an approximately close-to-far order. For each surface triangle f, we can compute $\mathbf{p}$'s Euclidean closest point on f, which we denote as $\mathbf{s}$. Then we can try to build a tetrahedral traverse from $\mathbf{s}$ to $\mathbf{p}$ using line 18. Once we successfully find a tetrahedral traverse, there is no need to look for a surface point further that $|\mathbf{s} - \mathbf{p}|$. Thus we can reduce the query radius to be $|\mathbf{s} - \mathbf{p}|$ every time we find a valid tetrahedral traverse for a surface point $\mathbf{s}$ that has a smaller distance to $\mathbf{p}$ than the current query radius, and continue querying, see the line 9 of line 18. The query will stop after all the surface elements within the query radius are examined. When the query stops, the last surface point that triggers the query radius update is the closest surface point to $\mathbf{p}$ that exists a tetrahedral traverse to allow $l(\mathbf{s}, \mathbf{p})$ to embedded to. According to Theorem 1 in the paper, $l(\mathbf{s}, \mathbf{p})$ is the $\mathbf{p}$'s shortest path to boundary.

When Ψ is causing no inversions and non-degenerate, line 18 is guaranteed to find the shortest path to the surface in a tetrahedral mesh with self-intersection.

1.2 Implementation and Acceleration

We have two implementations of the tetrahedral traverse algorithm: the static version and the dynamic version. In the static version of the implementation, the candidate intersecting face stack F, the candidate next tetrahedron stack T, and the traversed tetrahedra list $\mathcal{T}$ are all aligned static array on stack memory, which best utilizes the cache and SMID instructions of the processor. Of course, the static version of the algorithm will fail when the members in those arrays have exceeded their capacity. In those cases, the dynamic version will act as a fail-safe mechanism. The dynamic version of the traverse algorithm supports dynamic memory allocation thus those arrays can change size on the runtime. By carefully choosing the size of those static arrays, we can make sure most of the tetrahedral traverse is handled by the static implementation, optimizing efficiency and memory usage.

We also add some acceleration tricks to the implementations. First, we find that all the infinite loops encountered by the tetrahedral

ALGORITHM 4: FeasibleRegionCheck

```
Input: A surface point $\mathbf{s} \in M$, the closest type of $\mathbf{s}$, an interior point $\mathbf{p}$.
Output: Boolen value representing whether there $\mathbf{p}$ is at $\mathbf{s}$'s feasible region.
switch ClosestType do
    case At the interior do
        return True;
    end
    case On an edge do
        find two endpoints of the edge: $\mathbf{a}, \mathbf{b}$;
        find two neighbor faces of the edge: $f_1, f_2$;
        if $(\mathbf{p}-\mathbf{a})(\mathbf{b}-\mathbf{a}) < \epsilon_r$ then
            return False;
        else if $(\mathbf{p}-\mathbf{b})(\mathbf{a}-\mathbf{b}) < \epsilon_r$ then
            return False;
        $\mathbf{n}_1 = \text{normal}(f_1)$;
        $\mathbf{n}_1^{\perp} = (\mathbf{b}-\mathbf{a}) \times \mathbf{n}_1$;
        if $(\mathbf{p}-\mathbf{a})\mathbf{n}_1^{\perp} < \epsilon_r$ then
            return False;
        $\mathbf{n}_2 = \text{normal}(f_2)$;
        $\mathbf{n}_2^{\perp} = (\mathbf{a}-\mathbf{b}) \times \mathbf{n}_2$;
        if $(\mathbf{p}-\mathbf{a})\mathbf{n}_1^{\perp} < \epsilon_r$ then
            return False;
    end
    case Vertex do
        for $\mathbf{a} \in N_s(\mathbf{s})$ do
            if $(\mathbf{p}-\mathbf{s})(\mathbf{s}-\mathbf{a}) < \epsilon_r$ then
                return False;
        end
    end
    return True;
end
```

traverse only contain a few tetrahedra. Actually, the size of the loop is unbounded by the max number of tetrahedra adjacent to a vertex/edge. In practice, the number is even much smaller than that. Thus it is unnecessary to keep all the traversed tetrahedra in an array and check them at every step for the loop. Instead, we only keep the newest 16 traversed tetrahedra. In all our experiments, we have never encountered a loop with a size larger than 16. $\mathcal{T}$ is implemented as a circular array and the oldest member will be automatically overwritten by the newest member. The array of size 16 can be efficiently examined by the SIMD instructions. This modification reduced about 10% of the running time of our method. In addition, in the presence of inversions, we will stop the ray after it has passed 2 times of the distance between $\mathbf{p}$ and $\mathbf{s}$, instead of waiting for it to reach the boundary. This procedure reduced 20% of our running time.

We would like to point out that the shortest path query for each penetrated point is embarrassingly parallel because the process can be done completely independently. All the querying threads will share the BVH structure of the surface and the topological structure of the tetrahedral mesh. During the execution no modification to those data is needed, thus no communication is needed between

ALGORITHM 5: One XPBD Time Step with Collision Handling

```
Input: Position of the previous step: $\mathbf{x}_0$, velocity of the the previous step: $\mathbf{v}_0$, external force $\mathbf{f}$.
Output: Position $\mathbf{x}$ and $\mathbf{v}$ velocity of the new time step.
update $\mathcal{P}_{\text{in}}$ and $\mathcal{P}_{\text{out}}$ by DiscreteCollisionDetection;
$\mathbf{v} = \mathbf{v}_0 + \Delta t \mathbf{f}$ ;
$\mathbf{x} = \mathbf{x}_0 + \Delta t \mathbf{v}$ ;
XPBD material solve ;
$C = \emptyset$ ;
do in parallel
    for $\mathbf{s} \in \mathcal{P}_{in}$ do
        if DiscreteCollisionDetection($\mathbf{s}$) then
            $\mathbf{p}$ = $\mathbf{s}$'s overlapping interior point;
            $\mathbf{s}_{\text{closest}}$ = ShortestPathToSurface($\mathbf{p}$, $\mathbf{s}$);
            $C$.add(CollisionConstraint($\mathbf{s}$, $\mathbf{s}_{\text{closest}}$)) ;
    end
    for $\mathbf{s} \in \mathcal{P}_{out}$ do
        if ContinuousCollisionDetection($\mathbf{s}$) then
            $\mathbf{s}_{\text{collide}}$ = the surface point $\mathbf{s}$ collide with;
            $C$.add(CollisionConstraint($\mathbf{s}$, $\mathbf{s}_{\text{collide}}$)) ;
    end
end
for $c \in C$ do
    project constraint c;
end
$\mathbf{v} = (\mathbf{x} - \mathbf{x}_0)/\Delta t$
```

those threads. Also, our shortest path querying algorithm, especially the version with static arrays, can be efficiently executed on GPU.

1.3 Collision Handling Framework

We provide the pseudocode of our XPBD framework (see line 22) and our implicit Euler framework line 22. At the beginning of both frameworks, we separate all the surface points (edges can also be included) into two categories: initially penetrated points $\mathcal{P}_{\text{in}}$ and initially penetration-free points $\mathcal{P}_{\text{out}}$. For the XPBD framework, we apply collision constraints at the end of each time step, after the object has been moved by the material and the external force solving. Thus another DCD must be applied to every point in $\mathcal{P}_{\text{in}}$ to re-detect the tetrahedra including it for the purpose of shortest path query. In the implicit Euler framework, we only need to do DCD once because the collisions are built into the system as a penalty force and solved along with other forces. We use friction power that points to the opposite direction of the velocity and is proportional to the contact force.

Note that when solving models with existing significant self-intersections, we turn off CCD and set $\mathcal{P}_{\text{out}} = \emptyset$. $\mathcal{P}_{\text{in}}$ will not only contains surface points, but also all the interior vertices and all tetrahedra's centroids to resolve the intersection in the completely overlapping parts.

2 PROOF OF THEOREMS

2.1 Theorem 1

We first prove this lemma:

ALGORITHM 6: One Implicit Euler Time Step with Collision Handling

```
Input: Position of the previous step: $\mathbf{x}_0$, velocity of the the previous
       step: $\mathbf{v}_0$, external force $\mathbf{f}$.
Output: Position $\mathbf{x}$ and $\mathbf{v}$ velocity of the new time step.
update $\mathcal{P}_{\text{in}}$ and $\mathcal{P}_{\text{out}}$ by DiscreteCollisionDetection;
$C = \emptyset$ ;
do in parallel
    for $\mathbf{s} \in \mathcal{P}_{in}$ do
        $\mathbf{p}$ = $\mathbf{s}$'s overlapping interior point;
        $\mathbf{s}_{\text{closest}}$ = ShortestPathToSurface($\mathbf{p}$, $\mathbf{s}$);
        $C$.add(CollisionConstraint($\mathbf{s}$, $\mathbf{s}_{\text{closest}}$)) ;
    end
    for $\mathbf{s} \in \mathcal{P}_{out}$ do
        if ContinuousCollisionDetection($\mathbf{s}$) then
            $\mathbf{s}_{\text{collide}}$ = the surface point $\mathbf{s}$ collide with;
            $C$.add(CollisionConstraint($\mathbf{s}$, $\mathbf{s}_{\text{collide}}$)) ;
    end
end
Evaluate penetration force and its Jacobian (i.e. the Hessian of the
  penetration energy) ;
Evaluate internal force and its Jacobian (i.e. the Hessian of the
  elasticity energy);
Evaluate external forces ;
Build and solve implicit Euler system to obtain $\mathbf{x}$ and $\mathbf{v}$ [Baraff and
  Witkin 1998] ;
```

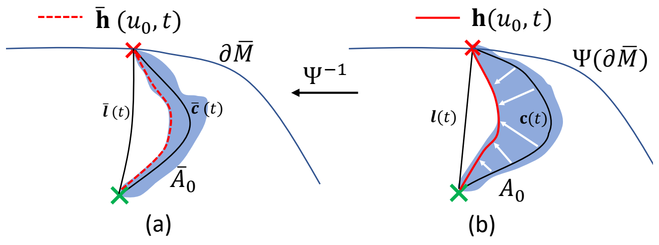

Fig. 2. *Constructing the undeformed pose curve. (a) The undeformed pose. (b) The deformed pose.*

Lemma 1. *For any point* $\mathbf{p} \in M$*, its shortest path to the boundary as* $\bar{\mathbf{p}}$ *is a line segment.*

Proof. Let's assume that $\mathbf{p}$'s shortest path to boundary as $\bar{\mathbf{p}}$ is not a line segment, and $\mathbf{p}$'s closest boundary point as as $\bar{\mathbf{p}}$ is $\mathbf{s}$ (as $\bar{\mathbf{s}}$). Then there must be a curve on the undeformed pose: $\bar{\mathbf{c}}(t) : I \mapsto \bar{M}, \bar{\mathbf{c}}(0) = \bar{\mathbf{p}}, \bar{\mathbf{c}}(1) = \bar{\mathbf{s}}$, s.t., $\mathbf{c}(t) = \Psi(\bar{\mathbf{c}}(t))$ is $\mathbf{p}$'s shortest path to boundary.

Since $\nabla\Psi > 0$, we know that Ψ is locally bijective. Thus, according to Heine–Borel theorem [Borel 1895], we can have a limited number of open sets $\mathcal{A}$ covering $\bar{M}$, such that Ψ is bijective on each open set in $\mathcal{A}$. We can select a subset $\mathcal{A}_0 \subseteq \mathcal{A}$, such that $\bar{\mathbf{c}}(I)$ is totally contained by $\bar{A}_0$= the union of $\mathcal{A}_0$, see Figure 2a.

We then construct a cluster of curves: $\mathbf{h}(u,t) = u\mathbf{c}(t) + (1-u)\mathbf{l}(t) : I \times I \mapsto \bar{M}$, such that, $\mathbf{h}(0,t) = \mathbf{c}(t), \mathbf{h}(1,t) = \mathbf{l}(t), \forall t \in I$, where $\mathbf{l}(t)$ is the line segment from $\mathbf{p}$ to $\mathbf{s}$, see Figure 2b. $\mathbf{h}(u,t)$ will smoothly deformed from $\mathbf{c}(t)$ to $\mathbf{l}(t)$ as u changes from 0 to 1. Note that any moment u, the length of the curve: $\mathbf{c}_u(t) = \mathbf{h}(u,\cdot)$ must be shorter than $\mathbf{c}(t)$.

Since $\mathbf{c}(I) \in A_0 = \Psi(\bar{A}_0)$, there must exist a $u_0 > 0$, such that, $\mathbf{h}([0,u_0], I) \in A_0$. Additionaly, because Ψ is bijective on A_0, we can define a undeformed pose curve cluster: $\bar{\mathbf{h}}(u,t) = \Psi^{-1}(\mathbf{h}(u,t)) : [0,u_0] \times I \mapsto \bar{A}_0$, as shown in Figure 2a.

We can then select another group open set $\mathcal{A}_1$ containing $\bar{\mathbf{h}}(u_0, I)$, and repeat the above procedure. This will give us another cluster of curves: $\bar{\mathbf{h}}(u,t) = \Psi^{-1}(\mathbf{h}(u,t)) : [u_0,u_1] \times I \mapsto \bar{A}_0$. During such process, the curve will not touch the boundary of the model, otherwise, there will be a shorter curve connecting $\bar{\mathbf{p}}$ and the boundary, which violates our assumption. Since $\mathcal{A}$ is a limited set, we can eventually obtain a $\bar{\mathbf{h}}(u,t) : [u_k, 1] \times I \mapsto \bar{M}$ within a limited $k+1$ steps, such that $\Psi(\bar{\mathbf{h}}(1,t)) = \mathbf{l}(t), \forall t \in I, \bar{\mathbf{h}}(1,0) = \bar{\mathbf{p}}$ and $\bar{\mathbf{h}}(1,1) = \bar{\mathbf{s}}$.

Here we have proved that $\mathbf{l}(t)$ is a valid path, which must be shorter than $\mathbf{c}(t)$ due to Euclidean metrics. Hence creating a contradiction. □

With Lemma 1, the proof of Theorem 1 becomes trivial. Since the shortest path to the boundary must be a valid path, of course it should be the *shortest* valid line segment to boundary.

2.2 Theorem 2

The proof of Theorem 2 is similar to Theorem 1. Say the closest boundary point is $\mathbf{s}' \in f$ (as $\bar{\mathbf{s}}'$) and the Euclidean closest boundary point on f is $\mathbf{s}$ (as $\bar{\mathbf{s}}$), where f is a boundary face. We also construct a cluster of curves: $\mathbf{h}(u,t) = u\mathbf{l}(t) + (1-u)\mathbf{l}'(t) I \times I \mapsto \bar{M}$, where $\mathbf{l}'(t)$ and $\mathbf{l}(t)$ are the line segment from $\mathbf{p}$ to $\mathbf{s}'$ and $\mathbf{s}$, respectively. This $\mathbf{h}(u,t)$ also holds the property that at any given moment u, the length of the curve: $\mathbf{c}_u(t) = \mathbf{h}(u,\cdot)$ must be shorter than $\mathbf{l}'(t)$.

Note that instead of fixing two ends, we only fix one end of $\mathbf{h}(u,t)$, as it deforms from $\mathbf{l}'(t)$ to $\mathbf{l}(t)$. Because Ψ is bijective on each boundary face, we can explicitly construct the line segment on the undeformed pose that goes from $\bar{\mathbf{s}}'$ and $\bar{\mathbf{s}}$, this allows us to move the position of the end point.

Similar to Lemma 1, we can induce a cluster of curves on the undeformed pose: $\bar{\mathbf{h}}(u,t)$, which will give us the pre-image of $\mathbf{l}(t)$ as $\bar{\mathbf{h}}(1,t)$, connecting $\bar{\mathbf{p}}$ and $\bar{\mathbf{s}}$. Hence we have proven that $\mathbf{l}(t)$ is also a valid path from $\bar{\mathbf{p}}$ to $\bar{\mathbf{s}}$. This contradicts the assumption that $\mathbf{s}'$ is the closest boundary point.

2.3 Element Traverse and Valid Path

We can give an equivalent definition of a curve being a valid path in the discrete case.

Theorem Supplementary 1. *A line segment connecting* $\mathbf{a} \in e_a$ *and* $\mathbf{b} \in e_b$ *in a mesh, is a valid path if and only it is included by element traverse from* e_a *to* e_b.

Proof. *Sufficiency.* If there exists such a element traverse $\mathcal{T}(\mathbf{a},\mathbf{b}) = (e_1, e_2, e_3, \ldots, e_{k-1}, e_k)$, s.t., $e_0 = e_a$ and $e_k = e_b$, we can explicitly construct a continuous piece-wise linear curve $\bar{\mathbf{c}}(t)$ defined on them, whose image is $\mathbf{c}(t)$. We do this by making a division of I: $I = [t_0,t_1] \cup [t_1,t_2] \cup [t_2,t_3] \cup \cdots \cup [t_{k-1},t_k]$, where $t_0 = 0, t_k = 1, t_0 \le t_1 \le t_2 \le \cdots \le t_k$. The division can be obtained by making $\frac{t_i}{t_{i+1}} = \frac{|\bar{\mathbf{r}}_i - \bar{\mathbf{r}}_{\bar{i}+1}|}{|\bar{\mathbf{r}}_{i+1} - \bar{\mathbf{r}}_{i+2}|}, \forall i = 1,2,\ldots,k+1$, where $\bar{\mathbf{r}}_i = \Psi|_{e_i}^{-1}(\mathbf{r}_i)$

is the preimage of the line segment's exit point from e_i. The curve can be constructed as:

$$\bar{\mathbf{c}}(t) = \frac{t - t_i}{t_{i+1} - t_i}\bar{\mathbf{r}}_i + (1 - \frac{t - t_i}{t_{i+1} - t_i})\bar{\mathbf{r}}_{i+1}, \text{if } t \in [t_i, t_{i+1}] \tag{1}$$

Necessity. Suppose we have a curve on the undeformed pose $\bar{\mathbf{c}}(t)$ connecting $\bar{\mathbf{a}}, \bar{\mathbf{b}}$, whose image under Ψ is a line segment. If $\bar{\mathbf{c}}(t)$ passes no vertex of $\overline{M}$, we directly obtain an element traversal by enumerating the elements that $\bar{\mathbf{c}}(t)$ passes by as t continuously changes from 0 to 1.

When $\bar{\mathbf{c}}(t)$ passes a vertex $\bar{\mathbf{v}}$ of $\overline{M}$, assume it goes from e_i to e_{i+1} at that point. According to the definition of a manifold, we can search around that $\bar{\mathbf{v}}$ and guarantee to have an element traversal from e_i to e_{i+1} formed by elements adjacent to $\bar{\mathbf{v}}$.

The case of $\bar{\mathbf{c}}(t)$ passing an edge can be proved similarly. □

2.4 Further Discussion on Inverted Elements

As we can see from Figure 6b of the paper, the line segment **sp** is only a subset of such a path constructed by our algorithm. In fact, in this case, the length of path $\mathbf{c}(t)$ is evaluated by this formula:

$$\int_0^1 sign((\mathbf{s} - \mathbf{p})r'(t)|r'(t)|dt \tag{2}$$

which means, in the presence of inverted tetrahedra, that the length of $\mathbf{c}(t)$ grows as it goes in the direction of $\mathbf{s}-\mathbf{p}$, and decreases if it goes in the opposite direction, which happens when it passes through the inverted tetrahedron. This is understandable because when solving the self-intersection, the penetrated point does not need to go back and forth, it only needs to pass through the overlapping part once. Thus the length of the overlapping part should only count once. With inverted tetrahedra, line 18 is actually constructing the shortest line segment connecting **p** and a surface point under the metrics introduced by Eq.2.

REFERENCES

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

David Baraff and Andrew Witkin. 1998. Large steps in cloth simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. 43–54.

Émile Borel. 1895. Sur quelques points de la théorie des fonctions. In *Annales scientifiques de l'École normale supérieure*, Vol. 12. 9–55.

Tom Duff, James Burgess, Per Christensen, Christophe Hery, Andrew Kensler, Max Liani, and Ryusuke Villemin. 2017. Building an orthonormal basis, revisited. *JCGT* 6, 1 (2017).