Exact and Efficient Polyhedral Envelope Containment Check

BOLUN WANG, Beihang University, China and New York University, USA
TESEO SCHNEIDER, New York University, USA
YIXIN HU, New York University, USA
MARCO ATTENE, Italian National Research Council, Italy
DANIELE PANOZZO, New York University, USA

We introduce a new technique to check containment of a triangle within an envelope built around a given triangle mesh. While existing methods conservatively check containment within a Euclidean envelope, our approach makes use of a non-Euclidean envelope where containment can be checked both *exactly* and *efficiently*. Exactness is crucial to address major robustness issues in existing geometry processing algorithms, which we demonstrate by integrating our technique in two surface triangle remeshing algorithms and a volumetric tetrahedral meshing algorithm. We provide a quantitative comparison of our method and alternative algorithms, showing that our solution, in addition to being exact, is also more efficient. Indeed, while containment within large envelopes can be checked in a comparable time, we show that our algorithm outperforms alternative methods when the envelope becomes thin.

ACM Reference Format:
Bolun Wang, Teseo Schneider, Yixin Hu, Marco Attene, and Daniele Panozzo. 2020. Exact and Efficient Polyhedral Envelope Containment Check. *ACM Trans. Graph.* 39, 4, Article 1 (July 2020), 14 pages. https://doi.org/10.1145/3386569.3392426

1 INTRODUCTION

The computation of distances between surfaces is a basic building block in geometry processing. In particular, the computation of the Hausdorff distance between an individual triangle $\mathcal{T}$ and a triangle mesh $\mathcal{M}$ is often used by meshing and remeshing algorithms (e.g., [Cheng et al. 2019; Hu et al. 2020, 2018]) to ensure geometric preservation up to a small distance ϵ. This distance allows algorithms to smooth out small details, fill small gaps, remove noise, and perform other operations to generate a high quality mesh, while at the same time bounding the geometrical approximation error. This bound is used, for example, in graphics applications to ensure sub pixels accuracy, or in finite element analysis to bound the error on the solution.

The Euclidean ϵ-envelope is the space of all points whose L^2 distance from a reference surface is less than ϵ (Figure 2). While checking if a point is contained within the envelope is a simple task, checking if an edge or triangle is contained within the envelope is a challenging problem, despite its apparent simplicity.

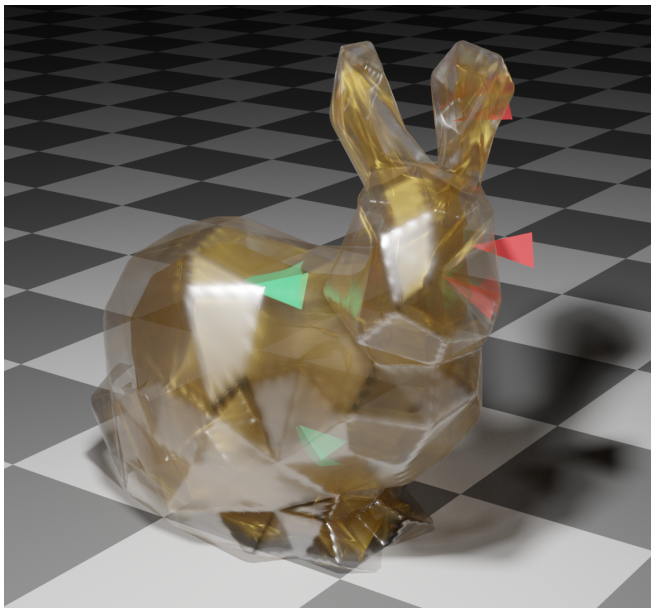


Fig. 1. Our method *exactly* detects if a triangle is inside (green) or outside (red) of an envelope (glass shell) of a bunny model (bronze).

Many existing algorithms in the literature perform this operation *inexactly* (e.g. by sampling the triangles), whereas just a few can be implemented exactly. A major limitation of inexact checks is that the running time (and memory usage) depends on ϵ: a thinner envelope will require more computations (e.g., more sampling points, larger number of refinements) to compensate for inaccuracy. This fact makes inexact checks impracticable (in terms of both memory and running time) for thin envelopes (Figure 23).

Additionally, while an inexact check is sufficient for certain applications, we discovered that it is problematic when used for remeshing. Remeshing algorithms use the envelope check during local operations, preventing any operation that will move the tracked surface outside of the envelope. Thus, these algorithms are based on a strong invariant as they assume that all the triangles remain inside the envelope. An inexact check leads to a subtle, yet major, problem: a valid triangle, "completely contained" within the envelope according to the inaccurate check, might be subdivided during the remeshing and, while its subtriangles are supposed to be in the envelope by construction, an actual check could reveal them

Authors' addresses: Bolun Wang, Beihang University, China, New York University, USA, wangbolun@buaa.edu.cn; Teseo Schneider, New York University, USA, teseo.schneider@nyu.edu; Yixin Hu, New York University, USA, yixin.hu@nyu.edu; Marco Attene, Italian National Research Council, Italy, marco.attene@ge.imati.cnr.it; Daniele Panozzo, New York University, USA, panozzo@nyu.edu.

0730-0301/2020/7-ART1 $15.00
https://doi.org/10.1145/3386569.3392426

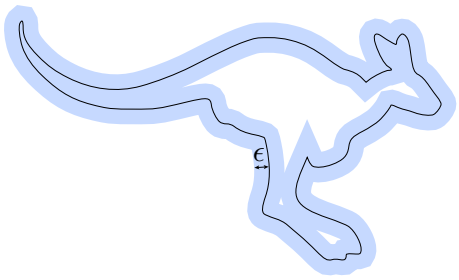


Fig. 2. A 2D curve in black (kangaroo shape) and its ϵ-envelope in blue.

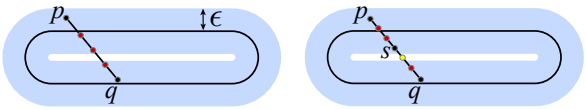


Fig. 3. Inexact checks may break algorithmic invariants in remeshing algorithms (e.g. triangles are inside the envelope at any stage). In this 2D example, a segment is declared to be *inside* if, after having sampled it, all its sample points are inside. This inexact check states that segment $[p, q]$ is inside (left), and the algorithm assumes that any of its sub-segments is also inside. For example, the algorithm may split $[p, q]$ at its midpoint s (right) with no need to further check. However, the subsegment $[s, q]$ is now outside even according to the inexact check, and this puts the algorithm in an inconsistent state as its invariant is violated. In the best case the algorithm may detect it, but the the $[s, q]$ would remain *locked* and cause over-refinement.

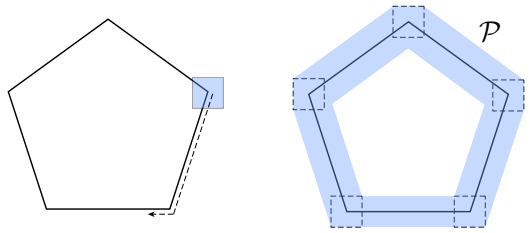


Fig. 4. 2D example of Minkowski polyhedral envelope $\mathcal{P}$ (in blue, right) of a 2D curve (in black, left) constructed by sweeping a square along the boundary of the curve.

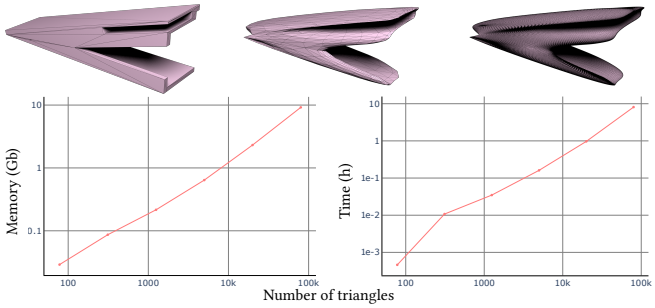


Fig. 5. Memory (left) and time (right) required to build a Minkowsky polyhedral envelope using CGAL. We start from a simple mesh with 78 triangles and refine it using Loop subdivision to create a sequence of increasingly dense meshes (top).

as outside. In the best case, if the algorithm does not crash due to a violation of its invariant, these triangles are *locked* in place and practically block mesh optimization in its surroundings, typically leading to over-refinement (Figure 22). We refer to Figure 3 and Appendix A for more details.

We note that it would be possible to adapt existing meshing algorithms to be robust with a non-exact envelope, but this will require changes on the application side, increasing the implementation complexity of the application. For instance, one can modify remeshing algorithms to unlock problematic triangles using heuristics. Tackling the low-level problem of envelope checking exactly concentrates the critical code in one place, which is easier to verify, thus downstream applications will not have to handle inconsistencies in the check, making them simpler to implement and more robust. While we see benefits in both approaches (the former one might lead to higher efficiency for a specific application, while the latter is more widely usable), we favor the second one since our long term goals is to have a broad range of robust geometric algorithms directly usable in different applications, without having to adapt the applications to handle special cases. Section 6 shows several applications of our method, where the integration is seamless.

Though a few methods exist that can be implemented exactly, such an exact implementation would require the use of arbitrary precision arithmetic that makes these methods too slow for most practical applications.

To avoid inaccuracy and performance loss at thin envelopes, we propose to design a predicate to check *exactly* if a triangle is contained within a polyhedral envelope $\mathcal{P}$, that is by itself contained within the ϵ-envelope. In this way, we obtain a conservative check ensuring the desired geometric tolerance, but we avoid the performance and locking problems.

This formulation has a simple, but impractical, exact solution: we can build the polyhedral envelope as the Minkowsky sum of a cube and the given triangle mesh using rational coordinates [Hachenberger 2009] (Figure 4), and check containment by computing intersections of the triangles with the discrete envelope. Unfortunately this approach does not scale well with the size of the mesh, and becomes unpractical (Figure 5) in terms of both construction time and memory usage for meshes with a few tens of thousand of triangles.

Our approach (Section 3) represents the polyhedral envelope $\mathcal{P}$ implicitly, sidestepping the onerous explicit construction, while still providing an exact evaluation using novel and efficient geometric predicates implemented using arithmetic filtering and floating point expansions to provide high performances. More precisely, $\mathcal{P}$ is represented as the union of a set of convex polyhedra, one for each triangle of $\mathcal{M}$, defined implicitly by a collection of half-spaces. To check for containment of the query triangle $\mathcal{T}$, we first check for intersections with each individual polyhedron, and then efficiently compute the union by implicitly checking for containment of the intersections. The resulting algorithm is exact, enabling us to design different polyhedral envelopes depending on the application,

and it is efficient, being comparable (and even much faster for thin envelopes) with inexact state of the art methods.

We quantitatively evaluate our algorithm and compare it with alternatives on a large set of benchmark problems (Section 5). We also integrate it with two surface remeshing algorithms [Cheng et al. 2019] and QSlim [Garland and Heckbert 1997] (Section 6.1) and a tetrahedral meshing algorithm [Hu et al. 2020] (Section 6.2), to show how it performs in real applications, and to demonstrate the benefits of replacing a conservative envelope check with our exact version. The reference implementation, the data, and the scripts to reproduce the results in the paper are provided in the additional material and are released as an open-source project https://github.com/wangbolun300/fast-envelope.

2 RELATED WORK

We first review the state of the art algorithms for envelope containment checks (Section 2.1). We then briefly summarize methods solving a closely related task, the minimization of the distance between two meshes (Section 2.2). Finally, we review applications requiring envelopes (Section 2.3) and the geometric predicate constructions used in our algorithm (Section 2.4).

2.1 Envelope for Geometric Error Checks

Previous works use an implicit or explicit envelope for geometry preservation during the shape approximation process.

Implicit Envelope. Implicit methods compute and bound the Hausdorff distance [Atallah 1983] from the output boundary to the input boundary to be smaller than a certain threshold. However, directly computing the exact Hausdorff distance is expensive [Barton et al. 2010]. To improve efficiency, some methods compute approximations using surface sampling [Cheng et al. 2019; Cignoni et al. 1996; Hu et al. 2017]. Those methods sample the surfaces and use point-to-surface distances to approximate the surface-to-surface distance; while efficient and simple, this method introduces an approximation error. Based on this strategy, Hu et al. [2019, 2018] then proposed a conservative way to check if a triangle is contained within an envelope with the sampling error compensated. The drawback of sampling-based methods is that a smaller envelope requires a higher density of sampling, making them unpractical for small envelopes (Figure 14). An alternative to the sampling is the derivation of upper bounds on the Hausdorff distance: Borouchaki and Frey [2005] controls the upper bound of Hausdorff distance in a local region, which is specifically designed to support local remeshing operations. Tang et al. [2009] compute both the lower bound and upper bound of Hausdorff distance between a triangle and a surface and tighten the bounds by subdividing the query triangle. When a small envelope requires tight bounds, the algorithm needs excessive levels of subdivision and thus becomes slow.

Explicit Envelope. Explicit envelopes methods compute an envelope shell, a discrete representation of the boundary of the envelope around the input boundary, and use it to test containment of other primitives. The envelope shells can be constructed using Minkowski sums [Kaul and Rossignac 1992], or offsetting [Jung et al. 2003] as a special case. Cohen et al. [1996] proposed to use the generalization of surface offset as envelope shell. But this method does not work for boundaries with self-intersections. Minkowski sums use a solid to sweep along the boundary of another solid and the occupied volume of the sweeping path is called swept volume that can be used as an envelope shell. The swept volume of Minkowski sums can be either a polygonal superset that is a set of intersected geometries [Ghosh 1993; Kaul and Rossignac 1992], or a Boolean union of the superset [Campen and Kobbelt 2010b; Hachenberger 2009]. Building a polygonal superset only is pretty fast but computing its union could be several orders of magnitude slower [Campen and Kobbelt 2010b]. The method in [Hachenberger 2009] is costly because it compute the union of the superset and requires arbitrary precision arithmetics. Campen and Kobbelt [2010b] proposed a more efficient, but still exact, algorithm that improves the running time considerably, but still does not scale to large models. Actually, when using Minkowski sums for envelope shell, it is not necessary to compute the union of the supersets.

2.2 Optimization-Based Geometric Error

Some variational methods incorporate geometric errors in their energies. For example, Frey and Borouchaki [2003] discussed an *a posteriori* interpolation error estimate based on the Hessian of the surface and proposed a new geometric error estimate related to the local deformation of the surface. Hoppe [1996] evaluates and minimizes an error involving the distance of each point to the input boundary when optimizing the vertex positions for a surface mesh with fixed connectivity. Instead of using point-to-boundary distance, Garland and Heckbert [1997] proposed plane-based error quadrics for estimating the geometric error of a processed surface mesh based on the sum of squared distances of a vertex to its associated planes (in a mesh a vertex can be seen as the intersection of a set of planes). Our algorithm can be used as a filtering criterion in the line search of these methods to ensure a bounded geometric error.

2.3 Applications

The main application of a geometric envelope is bounding geometric error. Since envelopes can be defined around both 2D curves or 3D surfaces, they can be used both in 2D and 3D meshing algorithms. Implicit envelope checks are widely used in surface mesh simplification [Borouchaki and Frey 2005; Cignoni et al. 1996], surface mesh refinement and optimization [Cheng et al. 2019; Hu et al. 2017], and mesh generation [Fu et al. 2014; Hu et al. 2019, 2018]. Explicit envelope, while usually more expensive, can be used in the same application as implicit envelopes [Cohen et al. 1996]. Besides, they can also be used in applications like [Mandad et al. 2015] that takes an envelope-shell-like tolerance volume as input and generates an approximated shape inside this tolerance volume.

2.4 Geometric Predicates

Typical geometric predicates evaluate the sign of a homogeneous polynomial and give information about the configuration of their input. E.g., given three points a, b, and c on the Euclidean plane, the sign of the 2×2 determinant $|b - a; c - a|$ tells us whether the three points are collinear or if they form a left or a right turn. Calculating a determinant using floating point arithmetic may lead to an incorrect

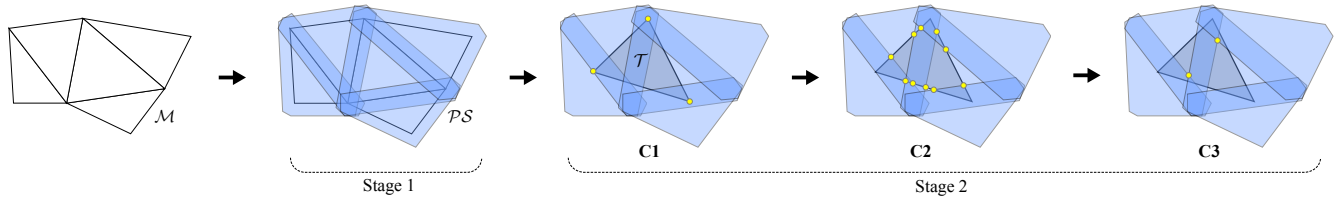


Fig. 6. Overview of our algorithm: starting from the input triangles soup $\mathcal{M}$, we build our polyhedra envelope $\mathcal{PS}$ (first stage), which we use to check if a query triangle $\mathcal{T}$ is inside using three checks between vertices, edges, and planes.

sign that, in turn, may easily put an algorithm in an inconsistent state, cause infinite loops, or even lead to a crash [Li et al. 2005]. Replacing floating point with arbitrary precision numbers [Fousse et al. 2007] solves the problem, but the slowdown is often unacceptable in downstream applications. That is why efficient predicate implementations use arithmetic filtering [Devillers and Pion 2003]: the polynomial is evaluated using floating point arithmetic, but we also estimate a bound for the rounding error. If the magnitude of the evaluated polynomial is smaller than the error bound, then its sign is uncertain (i.e. the filter *fails*), and the predicate is re-evaluated using arbitrary precision. The idea is that the failure rate is low enough to make the impact of arbitrary precision acceptable.

The error may be bounded based on the polynomial expression only (static filtering [Fortune and Van Wyk 1996]), or it may use the actual values of the input variables (dynamic filtering [Brönnimann et al. 1998]). For static filters, the error is pre-calculated and the runtime overhead is extremely low, but the failure rate is relatively high. Conversely, the error in dynamic filters is computed at each predicate call, thus leading to a higher overhead, but also to less failures. In an attempt to couple the advantages of both approaches, semi-static filtering splits the error in one static component to be pre-calculated, and one dynamic component that can be quickly computed at each call [Meyer and Pion 2008]. The approach in [Shewchuk 1997] falls in this latter category, though its floating point filtering is adaptively refined before reverting to arbitrary precision.

Though these approaches are both efficient and exact, correctness guarantees are lost if the predicate input is affected by an error. Thus, if *intermediate constructions* are used by a predicate, state of the art solutions rely on lazy exact evaluation [Pion and Fabri 2011]. Unfortunately, these solutions are far too slow when compared with floating point implementations.

Instead of relying on lazy exact evaluation, in this paper we rewrite standard predicates so that, if one of the input points needs to be derived as a composition of other values, such a derivation is included in the predicate itself. This allows to keep track of the roundoff and hence to implement filters enabling an efficient floating point calculation with guarantees. If the floating point filters fail, we do not directly switch to arbitrary precision (as done, e.g. in [Shewchuk 1997]). Instead, we re-evaluate the predicate using interval arithmetic, which reduces the overall need for arbitrary precision and thus improves performances.

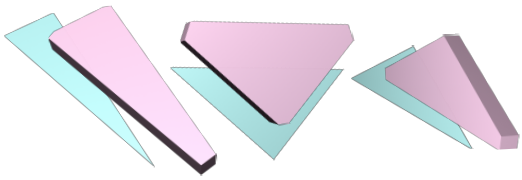


Fig. 7. Example of input triangles (blue) and their corresponding polyhedral envelope (pink).

3 METHOD

Our algorithm is composed of two stages (Figure 6): (1) convex polyhedral envelope construction (Section 3.1), and (2) containment check (Section 3.2). The first stage takes as input a 3D triangle soup $\mathcal{M}$ (i.e., a set of arbitrarily connected, potentially intersecting triangles with potentially shared vertices), and a user-controlled envelope size ϵ. It outputs a polyhedral envelope set $\mathcal{PS}$, containing *convex* polyhedral cells, one for each input triangle. The second stage takes as input the set $\mathcal{PS}$ and a *query* triangle $\mathcal{T}$, and calculates whether $\mathcal{T}$ is completely contained in $\mathcal{PS}$ *exactly*. More precisely, we check if all the points of $\mathcal{T}$ are in the interior of the polyhedra, and consider $\mathcal{T}$ to be outside if it has points *outside or on the boundary* of $\mathcal{PS}$.

The second stage could be realized by explicitly computing the union of all the convex polyhedra in $\mathcal{PS}$. However this is prohibitively costly and unnecessary; our algorithm is able to check for containment indirectly, without realizing the union of the polyhedra.

The shape of the convex polyhedral envelopes can be changed depending on the application. For instance, we can generate convex polyhedra to better approximate the L^2 envelope (Figure 18) or have polyhedra with different sizes, leading to adaptive envelopes (figures 16 and 17), depending on the application requirements.

3.1 Stage 1: Convex Polyhedron Creation

There exist several different ways to construct a convex polyhedron $\mathcal{P}$ containing an input triangle while being contained in an L^2-distance envelope. An easy construction is to use a Minkowsky sum on every input triangle with a approximation of a sphere (Section 5.3 shows an example), however it is slow and might lead to a potentially high number of faces, depending on the tessellation of the sphere.

We propose a different construction that strives to minimize the construction cost and the number of faces, since this will reduce

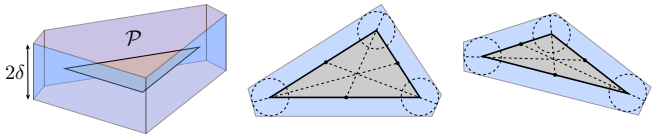


Fig. 8. 3D example of a convex polyhedron $\mathcal{P}$ (left) for a single triangle and its construction on the plane of the triangle. Middle, the case where two angles are less than 90 degree and one is 90 degree; right, the case that maximal angle is more than 90 degree.

the complexity of the next phase of the algorithm (Figure 7). Given a triangle $\mathcal{T}$ and a distance δ, we construct a plane $\mathcal{T}_{\text{floor}}$ below $\mathcal{T}$ and a plane $\mathcal{T}_{\text{ceil}}$ above it, so that both the planes are parallel to $\mathcal{T}$ and have distance δ from it. Also, we construct three planes $\mathcal{T}^i_{\text{side}}$, $i \in \{1, 2, 3\}$, each orthogonal to $\mathcal{T}$ and parallel at distance δ to one of its edges. $\mathcal{T}_{\text{floor}}$, $\mathcal{T}_{\text{ceil}}$ and the $\mathcal{T}_{\text{side}}$'s collectively bound a triangular prism containing $\mathcal{T}$. Though the prism contains $\mathcal{T}$, some of its points may be arbitrarily far from the triangle (e.g., when one of the angles of $\mathcal{T}$ may become acute). To avoid this problem and ensure a distance bound, for any acute vertex of $\mathcal{T}$ we cut the prism using an additional plane at distance δ from the vertex and orthogonal to the line ℓ connecting the vertex and the triangle's barycenter (Figure 8). This construction can be avoided when the angle becomes obtuse, since the distance is bounded in any case.

Proposition 3.1. *The polyhedron $\mathcal{P}$ obtained by offsetting a triangle $\mathcal{T}$ by $\delta = \epsilon/\sqrt{3}$ is convex and the distance between any point in $\mathcal{P}$ and the triangle is at most ϵ.*

Proof. $\mathcal{P}$ is convex because it is the intersection of half-spaces.

To show that the distance is bounded, we define ω to be the maximum distance between the triangle and $\mathcal{P}$ in the plane of the triangle; then the actual maximum distance (in 3D) is $d = \sqrt{\omega^2 + \delta^2}$ (Figure 9, left) which we will show that, for $\delta = \epsilon/\sqrt{3}$, is smaller than ϵ.

The maximum distance ω is attained from the vertices of $\mathcal{T}$ and $\mathcal{P}$. For every vertex we have two cases: the angle acute or right, or obtuse.

When the angle is acute (or right), the position of the orthogonal plane to ℓ depends on the barycenter of the triangle. The position of ℓ that maximizes ω is obtained in the limit when the barycenter is on one of the edges, that is when the line ℓ is one of the two edges (Figure 9, right). In this case $\omega = \sqrt{2}\delta$ which implies that $d = \sqrt{2\delta^2 + \delta^2} = \epsilon$ (Figure 9 middle).

For any angle larger than 90 degrees, the intersection point becomes closer (and δ in the limit) and therefore the distance shorter. □

Note that, while this construction ensures that a query triangle is within an ϵ-distance from the triangle soup $\mathcal{M}$, for large "flat" regions the check is conservative and the query triangle will leave the envelope for any distance greater than δ. For the example in Figure 18 the ratio between the volume of our envelope and the volume of an Euclidean L^2 envelope is approximately $\sqrt{3}$.

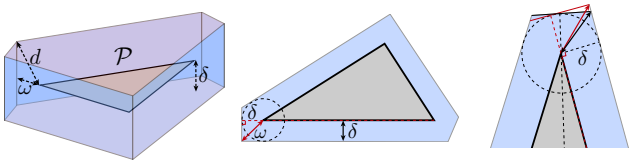


Fig. 9. Illustration for the bound on maximum distance. The dashed circle are at δ from the input triangle (in gray) creating the blue envelope. The red lines represent the maximum distance in the plane.

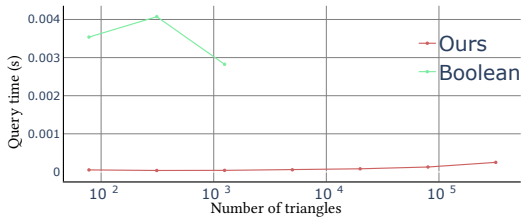


Fig. 10. Plot of query time versus number of faces for the model in Figure 5 using an envelope realized with the Boolean union of polyhedra and our method. Note that the explicit construction of the envelope is very expensive: 3 hours for 312 faces, 14 hours for 1248 faces faces. Our method avoids the explicit construction and it is faster at query time.

3.2 Stage 2: Envelope Check

Equipped with the envelope $\mathcal{PS}$, composed of the union of open convex polyhedra, one for each triangle of the input mesh, we can now present the algorithm to check for containment. We first describe our method assuming the use of exact arithmetic, then discuss the challenges of implementing it using floating point arithmetic, and finally we illustrate our efficient exact solution. Our algorithm requires two novel geometric predicates, which we detail in Section 4.

Algorithm Overview. Since our envelope $\mathcal{PS}$ is defined as the Boolean union of individual convex polyhedra $\mathcal{P}$, a triangle $\mathcal{T}$ is contained within $\mathcal{PS}$ if the Boolean subtraction of $\mathcal{PS}$ from $\mathcal{T}$ is empty:

$$\mathcal{T} \setminus (\mathcal{P}_1 \cup \mathcal{P}_2 \cup \ldots \cup \mathcal{P}_n) = \emptyset.$$

This remark leads to a possible simple solution of our problem: compute the union of $\mathcal{P}_i$ as a valid mesh and check if $\mathcal{T}$ is contained within. This naive approach suffers from similar limitations as an explicit Minkowsky sum (Figure 5): the construction of the envelope is extremely slow and computing containement in an exact Boolean union is also expensive (Figure 10).

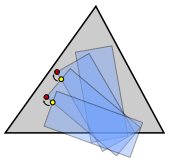
To avoid the computation of the explicit union, we can rewrite this expression as

$$\mathcal{T} \setminus \mathcal{P}_1 \setminus \mathcal{P}_2 \setminus \ldots \setminus \mathcal{P}_n = \emptyset, \tag{1}$$

which leads to a simple algorithm. Starting from $\mathcal{T}$, sequentially carve out the parts overlapping with any polyhedron $\mathcal{P}_i$ until you either have the empty set (and thus $\mathcal{T}$ is contained in $\mathcal{PS}$) or you "run out" of polyhedra (and thus $\mathcal{T}$ is not contained in $\mathcal{PS}$).

Floating Point Challenges. Implementing this algorithm using rational numbers (or infinite precision arithmetic) is straightforward

but has impractical runtime since every subtraction might double the size of the rational numbers used to store the coordinates, since the output of an operation is the input of the next (see inset).



We thus design an equivalent version of this algorithm tailored to avoiding cascading operations and implementable *exactly* using floating point arithmetic. While some of our algorithmic choices in the following section might seem exotic at a first glance, they are actually necessary to ensure both efficiency and exactness. In particular there are two tasks in the algorithm that are challenging to solve with floating point computations:

- If a polyhedron $\mathcal{P}$ has a facet with more than three vertices represented in floating-point coordinates, they will likely not be *exactly coplanar*, which might make the polyhedra concave [Si and Shewchuk 2014]. To avoid this problem, we propose to never realize $\mathcal{P}$ explicitly. Instead, we represent $\mathcal{P}$ as an intersection of half-spaces, each defined by three non-collinear points.
- The intersection between a triangle $\mathcal{T}$ and the boundary $\partial\mathcal{P}$ of a convex polyhedron $\mathcal{P}$ will usually not lie on $\partial\mathcal{P}$ or $\mathcal{T}$ because of rounding (Appendix B). This is problematic, since the intersection might be randomly either inside or outside $\mathcal{P}$. To prevent this problem, we introduce a custom predicate that avoids representing the intersection explicitly, exploiting the fact that $\mathcal{P}$ is defined as an intersection of half-spaces (sections 3.2.2 and 3.2.3).

Efficient Implementation. Our algorithm is summarized in Listing 1 and Figure 6 (Stage 2), and is based on the following theorem:

Theorem 3.2. *A triangle $\mathcal{T}$ is contained within $\mathcal{PS}$ (or equivalently Equation (1) holds) if and only if the following three conditions are true:*

C1 *the vertices v_i of $\mathcal{T}$ are inside $\mathcal{PS}$,*

C2 *the intersection point of the edges e_i of $\mathcal{T}$ with any facet of any polyhedron $\mathcal{P} \in \mathcal{PS}$ is contained in at least another polyhedron $\mathcal{P}^\star \neq \mathcal{P}$, with $\mathcal{P}^\star \in \mathcal{PS}$,*

C3 *for any pair of facets $F^l_{\mathcal{P}_i}, F^m_{\mathcal{P}_j}$ ($\mathcal{P}_i$ and $\mathcal{P}_j$ might be the same) the intersection point $\mathcal{T} \cap F^l_{\mathcal{P}_i} \cap F^m_{\mathcal{P}_j}$ is contained in at least another polyhedron $\mathcal{P}^\star$ with $\mathcal{P}^\star \neq \mathcal{P}_i, \mathcal{P}^\star \neq \mathcal{P}_j$, and $\mathcal{P}^\star \in \mathcal{PS}$.*

Proof. One direction of the implication is trivial, if $\mathcal{T}$ is inside the envelope, any point $\boldsymbol{p} \in \mathcal{T}$ is also inside. In particular its vertices (**C1**), the intersections of its edges with the facets of $\mathcal{P}$ (**C2**), or its intersections with two facets (**C3**).

The second direction can be proven by contradiction; let **C1**, **C2**, and **C3** be true and assume that $\mathcal{T}$ is outside the envelope, that is there exists a point $\boldsymbol{q} \in \mathcal{T}$ not in $\mathcal{PS}$.

Let us assume that $\boldsymbol{q}$ is one of the three vertices $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3$ of $\mathcal{T}$; this contradicts **C1**.

We now assume that $\boldsymbol{q} \in e_1$ (the other edges follow by re-enumeration), where e_1 is the open edge connecting $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$. Because of **C1**, there exists a polyhedron $\mathcal{P}^\star$ that contains $\boldsymbol{v}_1$. If $\mathcal{P}^\star$ contains also $\boldsymbol{v}_2$ (Figure 11 (a)), since $\mathcal{P}^\star$ is convex, the whole

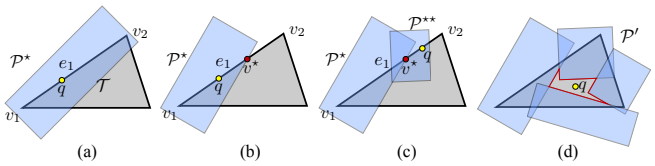


Fig. 11. Explanation of the different stages of the proof of Theorem 3.2

edge e_1 is contained in $\mathcal{P}^\star$ which contradicts the assumption. In the other case, $\boldsymbol{v}_2 \notin \mathcal{P}^\star$, there exists a point $\boldsymbol{v}^\star \neq \boldsymbol{v}_1$ (since $\mathcal{P}^\star$ is an open polyhedron) which is the intersection between e_1 and $\mathcal{P}^\star$. If $\boldsymbol{q} \in (\boldsymbol{v}_1, \boldsymbol{v}^\star)$, it contradicts the assumption (Figure 11 (b)). In the other case, $\boldsymbol{q} \in [\boldsymbol{v}^\star, \boldsymbol{v}_2)$, because of **C2** there exists another polyhedron $\mathcal{P}^{\star\star}$ that contains $\boldsymbol{v}^\star$ and, since $\mathcal{P}^{\star\star}$ is open, the intersection $\boldsymbol{v}^{\star\star}$ between $\mathcal{P}^{\star\star}$ and e_1 is different from $\boldsymbol{v}^\star$ Figure 11 (c)). In other words $\boldsymbol{q} \in (\boldsymbol{v}^{\star\star}, \boldsymbol{v}_2)$; we now repeat the reasoning for this new interval and shows that $\boldsymbol{q} \notin e_1$ since the number of polyhedra in $\mathcal{PS}$ is finite, which contradicts $\boldsymbol{q} \in e_1$.

The last case is that $\boldsymbol{q}$ is in the open triangle. We first remark that the result of the previous proof is a set $\mathcal{S}$ (blue in Figure 11 (d)) of polyhedra covering the edges of $\mathcal{T}$ and if $\boldsymbol{q} \in \mathcal{S}$ it contradicts the assumption. Thus, let $\boldsymbol{q} \in \mathcal{T} \setminus \mathcal{S}$; the boundary of $\mathcal{S}$ is a polygon (highlighted in Figure 11 (d)) whose vertices results as the intersection between two faces of polyhedra in $\mathcal{P}$. Any vertex of this polygon, from **C3**, is strictly inside another polyhedron $\mathcal{P}'$. If $\mathcal{P}'$ covers the whole polyhedron it contradicts the assumption. In the other case, it will shrink the area of $\mathcal{T} \setminus \mathcal{S}$ since it is open and will generate new points. We repeat this reasoning a finite number of time (since the set $\mathcal{PS}$ is finite) which implies that the area of the polygon can only be zero and therefore $\boldsymbol{q} \notin \mathcal{T}$, which contradicts the assumption. □

The algorithm and corresponding theorem have been designed to enable efficient verification using floating point predicates. We describe the algorithm first, and postpone the details of the implementation of the predicates to Section 4, but it is important to note now that this algorithm never explicitly requires divisions in the constructions, which is a mandatory requirement to derive filters using [Meyer and Pion 2008] and to exactly evaluate the predicates using floating point expansions.

Our algorithm requires checking the three conditions in Theorem 3.2 (sections 3.2.1, 3.2.2, and 3.2.3), the first can be realized using standard orientation predicates, while the other two will require custom predicates to guarantee correctness. We propose two versions of the algorithm. The first is a direct implementation of Theorem 3.2 (Listing 1), which is simple but computes many unnecessary implicit intersection points. The second is an accelerated version that discards unnecessary intersection tests (Section 3.2.4). Both algorithms are exact and they rely on three subroutines to verify if the three conditions of Theorem 3.2 hold for a given triangle and envelope, which we describe in the next section. We will discuss timings in more details in Section 5 but, as a reference, a naive implementation of the first algorithm using rational numbers is 100x slower than using our predicates, and it is 10 000x slower than the second version of our algorithm.

Listing 1. Overview of the three stages of our envelope check algorithm.

```
envelope_check(T, PS)
  //(1) v_i is contained in one of the PS, Section 3.2.1
  for (i = 1, 2, 3)
    out = point_out(v_i, PS)
    if(out == OUT) return OUT

  //(2) triangle edge intersection with F_P, Section 3.2.2
  for(P ∈ PS)
    for(F_P ∈ P)
      for (i = 1, 2, 3)
        out = edge_plane_out(e_i, F_P, PS \ P)
        if(out == OUT) return OUT

  //(3) intersection of T, F^l_{P_i}, and F^m_{P_j}, Section 3.2.3
  for(P_i, P_j ∈ PS)
    for(F^l_{P_i} ∈ P_i, F^m_{P_j} ∈ P_j)
      out = plane_plane_tri_out(T, F^l_{P_i}, F^m_{P_j}, PS \ (P_i ∪ P_j))
      if(out == OUT) return OUT

  return IN
```

3.2.1 ***C1**: Point in Polyhedra.* A vertex $\boldsymbol{v}_i$ of $\mathcal{T}$ is inside $\mathcal{P} \in \mathcal{PS}$ if $\boldsymbol{v}_i$ is inside all the halfspaces that define $\mathcal{P}$ or, equivalently, if $\boldsymbol{v}_i$ is *below* all the oriented planes $F_{\mathcal{P}}$ that define these halfspaces. Since the planes $F_{\mathcal{P}}$ are encoded as a triplet of points $(\boldsymbol{a}_F, \boldsymbol{b}_F, \boldsymbol{c}_F)$, determining whether $\boldsymbol{v}_i$ is in $\mathcal{P}$ reduces to evaluating a `orient3d`$(\boldsymbol{v}_i, \boldsymbol{a}_F, \boldsymbol{b}_F, \boldsymbol{c}_F)$ predicate for each plane. Standard efficient implementations exist to exactly evaluate this predicate [Lévy 2019; Shewchuk 1997], and are sufficient to evaluate condition **C1** (see Listing 2).

Listing 2. Overview of our point check, condition **C1**.

```
point_out(v_i, PS)
  for(P ∈ PS)
    counter = 0
    for(F_P ∈ P)
      ori = orient3d(v_i, F_P)
      if(ori == IN) counter++

    //if the point is inside for all faces of P
    //we found it
    if(counter == ||P||) return IN
  return OUT
```

3.2.2 ***C2**: Implicit Edge Facet Check.* A trivial way to evaluate our second condition could be explicitly computing the intersection point $\boldsymbol{p}$ and then using the approach described in Section 3.2.1. Unfortunately, this approach is not exact: the coordinates of $\boldsymbol{p}$ are not necessarily representable using floating point numbers, and the rounding error may be sufficiently large to make standard predicates return a wrong result (Appendix B).

We thus propose, instead of computing $\boldsymbol{p}$ as an explicit intersection of an edge and a facet, to represent it implicitly as a set of five points, $\boldsymbol{r}$ and $\boldsymbol{s}$ defining the edge, and $\boldsymbol{t}, \boldsymbol{u}, \boldsymbol{v}$ defining the facet. We call such an implicit point an *LPI point*, short for Line-Plane Intersection.

This change in perspective requires implementing a new 3D orientation predicate. Instead of using the coordinates of $\boldsymbol{p}$ to define it, we use the coordinates of the five points generating it. Thus, the input to our custom `orient3d_LPI` predicate (Section 4.1) is made of eight points, five defining the LPI point plus three defining the reference plane. To check for condition **C2**, we iterate over all the polyhedra and check the orientation of their planes with respect to the implicit intersection, Listing 3. Note that `orient3d_LPI` does not enforce that the point $\boldsymbol{p}$ is between $\boldsymbol{r}$ and $\boldsymbol{s}$. This additional check may be simply implemented using traditional orientation predicates: $\boldsymbol{p}$ is in the segment if $\boldsymbol{r}$ and $\boldsymbol{s}$ lie on opposite sides of the plane spanned by $\boldsymbol{t}, \boldsymbol{u}, \boldsymbol{v}$.

Listing 3. Overview of our implicit edge facet check, condition **C2**.

```
edge_plane_out(e_i, F_P, PS')
  //check if the endpoints are on opposite sides
  //of plane F_P and not on the plane
  ori1 = orient3d(v_{e_i^1}, F_P)
  ori2 = orient3d(v_{e_i^2}, F_P)
  if(ori1 == ori2 or ori1 == ON or ori2 == ON)
    return SKIP

  for(P' ∈ PS')
    counter = 0
    for(F_P' ∈ P')
      ori = orient3d_LPI(e_i, F_P, F_P')
      if(ori == IN) counter++
    //if the point is inside for all faces of P'
    //we found it
    if(counter == ||P'||) return IN

    //in the other case
    //we continue searching for another polyhedron

  return OUT
```

Evaluating our custom predicates amounts to calculate the sign of homogeneous polynomials, where we exploit state of the art tools to derive tight semi-static filters [Meyer and Pion 2008] for their floating point evaluation. When the semi-static filter fails, we re-evaluate the polynomial using interval arithmetic [Brönnimann et al. 1998]. If the resulting interval contains the zero (i.e., the dynamic filter fails) we evaluate the polynomial exactly using floating point expansions [Joldes et al. 2016]. For the sake of clarity, we postpone the description of these steps to Section 4.

3.2.3 ***C3** Implicit Triangle Facet-Facet Check.* This algorithm is similar to the segment plane intersection, but a Three-Planes Intersection (TPI) point is defined by three triplets of points, one for each plane. Our `orient3d_TPI` predicate uses the coordinates of twelve points, nine defining the TPI points (i.e., the three triplets) plus three defining the reference plane (Section 4.2). As for the line case, the predicate does not check for the intersection point $\boldsymbol{p}$ being inside the triangle. These check can be easily done with a series of `orient3d_TPI` with planes passing trough the edges of $\mathcal{T}$, Listing 4. We use this new predicate to check for condition **C3**. As for the line case, we implement semi-static filters with interval arithmetic and floating point expansions (Section 4).

Listing 4. Overview of our implicit facet facet triangle check, condition **C3**.

```
plane_plane_tri_out(T, F^l_{P_i}, F^m_{P_j}, PS')
  q = a point not on the plane of T
  counter = 0
  for (i = 1, 2, 3)
    F_i = plane passing trough v_i, v_{i+1}, and q
    ori = orient3d_TPI(T, F^l_{P_i}, F^m_{P_j}, F_i)
```

```
    if(ori == IN) counter++

  if(counter ≠ 3) return SKIP

  for(P' ∈ PS')
    counter = 0
    for(F_P' ∈ P')
      ori = orient3d_TPI(T, F^l_Pi, F^m_Pj, F_P')
      if(ori == IN) counter++

    //if the point is inside for all faces of P'
    //we found it
    if(counter == ||P'||) return IN

    //in the other case
    //we continue searching for another polyhedron

  return OUT
```

3.2.4 Acceleration. To reduce the running time of our algorithm we designed three strategies to limit the number of intersection calculations to only those which are really necessary. On average the three strategies give us 100x speedup.

In the first strategy, we filter the polyhedra intersecting triangle $\mathcal{T}$ using a conservative axis aligned bounding box tree [Lévy 2019]. This simple pass is extremely cheap and allows to prune unnecessary computations.

The second strategy consists of further removing unnecessary polyhedra using floating point arithmetic only, with no switch to interval or expansion arithmetic. After having checked condition **C1** for the three vertices of the triangle, we may have three possible cases: (1) at least one vertex is not in $\mathcal{PS}$; (2) there exists one $\mathcal{P} \in \mathcal{PS}$ that contains all the three vertices; (3) the three vertices are all in $\mathcal{PS}$, but in different $\mathcal{P}$s. In the first (resp. second) case, we simply stop the algorithm as the triangle is outside (resp. inside) the envelope. In the third case, the triangle must necessarily intersect the facets of polyhedra in $\mathcal{PS}$. All the polyhedra that do not have intersecting facets can be safely rejected. When checking for intersections, we use our predicates and reject polyhedra only if all the filters provide a guaranteed answer, that is we conservatively keep the pair in case of doubt.. This conservative filtering reduces the number of polyhedra by 30% from the rough selection coming from the bounding boxes, with negligible cost, since all the checks are done in floating-point only. In addition of reducing the number of polyhedra to be checked, it also provides with the list of possibly intersected facets for every polyhedron, which we use in our algorithm. Specifically, when computing **C2** and **C3**, we only iterate over possibly intersecting facets instead of all facets. We also observe that the intersecting facets are the ones "deciding" if the triangle is outside. In other words, when the triangle leaves the envelope it intersects one of the facets of a polyhedron. Using this observation, we change the order in which we check for orientation in the inner loop. That is, we compute the orientation of the implicit intersection point first against intersecting faces since they are the more likely to decide that the point is outside.

The third strategy is inspired from the proof of Theorem 3.2: we use a covering strategy to reduce the number of points generated and checked (Figure 12). We incrementally construct a *covering set* C. We start from a polyhedron $\mathcal{P}_C$ containing one of the vertices of $\mathcal{T}$, which we obtain from the check of **C1**, and add it to C. Then, we proceed by looking for a polyhedron $\mathcal{P}^\star \neq \mathcal{P}_C$ containing the intersection points $\boldsymbol{q}_i$ between $\mathcal{P}_C$ and the triangle's edges. The key ingredient is to first search for $\mathcal{P}^\star$ in C and then search in $\mathcal{PS}$. If $\mathcal{P}^\star \in C$, it means that $\boldsymbol{q}_i$ is already covered and there is no need for generating new intersection points (between $\mathcal{P}^\star$ and the edge of $\mathcal{T}$). In the other case ($\mathcal{P}^\star \notin C$), $\boldsymbol{q}_i$ is a new intersection, therefore we add $\mathcal{P}^\star$ to C and proceed with the new intersection points between $\mathcal{P}^\star$ and the edge. The algorithm terminates when either one of the intersections is outside or when any new point is contained in C. This strategy incrementally builds a covering of the edges of $\mathcal{T}$ while limiting the number of new intersection points, since any polyhedron in C never generates new points. Once the edges are covered, we follow a similar strategy for covering the interior of $\mathcal{T}$.

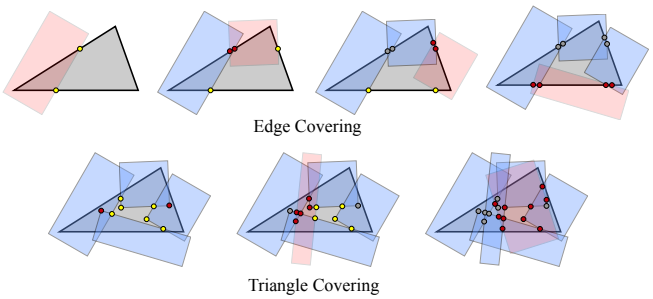


Fig. 12. Covering of a triangle by incrementally constructing the covering C (in blue), represented by the gray triangle. We start by selecting one polyhedron (in red), and generate its intersection points with $\mathcal{T}$. We classify the points into covered in C (red) and new points (yellow). For every uncovered point (yellow) we search for a polyhedron in $\mathcal{PS}$ (red in the second image) that covers it and insert it into C. We proceed until all edges are covered (top row), that is all new intersection points are covered in C (gray or red). We follow the same strategy for covering the interior, bottom row.

4 PREDICATES

To simplify the notations, we assume that the vectors are row vectors. We denote by the subscript x, y, and z the three components of vectors. We denote by $\times$ the cross product between two vectors and by $\cdot$ the dot product.

4.1 orient3d_LPI

To be able to build the predicates we first observe that the point $\boldsymbol{p}$ is the result of the intersection of a plane and a line (i.e., a facet against a triangle edge), whose numerical error cannot be bounded when computed using floating point arithmetic. However for building the predicate we are only interested to evaluate the sign of a polynomial, and its numerical inaccuracy can be bounded. Note, to be able to bound the inaccuracy we need to reformulate the predicate avoiding divisions.

Let $\boldsymbol{r}$ and $\boldsymbol{s}$ be two points defining a straight line $\mathcal{L}$. Also, let $\boldsymbol{t}$, $\boldsymbol{u}$, $\boldsymbol{v}$ be three points defining a plane $\mathcal{P}$, and $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$ be three points defining a reference plane. Assuming that $\mathcal{L}$ and $\mathcal{P}$ intersect at a single point $\boldsymbol{p}$, $\texttt{orient3d\_LPI}(\boldsymbol{r}, \boldsymbol{s}, \boldsymbol{t}, \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$ is the sign of the volume of tetrahedron $(\boldsymbol{p}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$.

The intersection point $\boldsymbol{p}$ is

$$\boldsymbol{p} = \boldsymbol{r} + \frac{\alpha}{\beta}(\boldsymbol{s} - \boldsymbol{r}), \quad \text{where } \alpha = \begin{vmatrix} \boldsymbol{r} - \boldsymbol{t} \\ \boldsymbol{u} - \boldsymbol{t} \\ \boldsymbol{v} - \boldsymbol{t} \end{vmatrix} \text{ and } \beta = \begin{vmatrix} \boldsymbol{r} - \boldsymbol{s} \\ \boldsymbol{u} - \boldsymbol{t} \\ \boldsymbol{v} - \boldsymbol{t} \end{vmatrix}.$$

If $\beta = 0$, $\boldsymbol{p}$ is undefined (i.e., $\mathcal{L}$ and $\mathcal{P}$ do not intersect at a single point). Otherwise, `orient3d`$(\boldsymbol{p}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$ is:

$$O(\boldsymbol{p}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}) = \begin{vmatrix} \boldsymbol{p} - \boldsymbol{c} \\ \boldsymbol{a} - \boldsymbol{c} \\ \boldsymbol{b} - \boldsymbol{c} \end{vmatrix} = \frac{1}{\beta^3} \begin{vmatrix} \beta\boldsymbol{p} - \beta\boldsymbol{c} \\ \beta\boldsymbol{a} - \beta\boldsymbol{c} \\ \beta\boldsymbol{b} - \beta\boldsymbol{c} \end{vmatrix} = \frac{1}{\beta^3} \begin{vmatrix} \beta\boldsymbol{r} + \alpha\boldsymbol{s} - \alpha\boldsymbol{r} - \beta\boldsymbol{c} \\ \beta\boldsymbol{a} - \beta\boldsymbol{c} \\ \beta\boldsymbol{b} - \beta\boldsymbol{c} \end{vmatrix}.$$

Note that this expression uses input values only (i.e., the intermediate construction $\boldsymbol{p}$ is not part of the expression). Furthermore, its sign can be obtained by composing the sign of β, which is a homogeneous polynomial, with the sign of

$$O^*(\boldsymbol{p}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}) = \beta^3 O(\boldsymbol{p}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}) = \begin{vmatrix} \beta\boldsymbol{r} + \alpha\boldsymbol{s} - \alpha\boldsymbol{r} - \beta\boldsymbol{c} \\ \beta\boldsymbol{a} - \beta\boldsymbol{c} \\ \beta\boldsymbol{b} - \beta\boldsymbol{c} \end{vmatrix}$$

which is another homogeneous polynomial. To summarize, the sign (and therefore the result) of our predicate is

$$\text{sign}(O) = \begin{cases} -\text{sign}(O^*) & \text{for } \beta < 0 \\ \text{sign}(O^*) & \text{otherwise} \end{cases}.$$

The semi-static filter, as calculated by [Meyer and Pion 2008], for the polynomial expression of β is

$$\begin{aligned} \varepsilon_\beta &= 5.1107127829973299\,10^{-15}\delta_1\delta_2\delta_3 \\ \delta_1 &= \max\{|r_x - s_x|, |u_x - t_x|, |v_x - t_x|\} \\ \delta_2 &= \max\{|r_y - s_y|, |u_y - t_y|, |v_y - t_y|\} \\ \delta_3 &= \max\{|r_z - s_z|, |u_z - t_z|, |v_z - t_z|\}. \end{aligned}$$

This means that, if β is calculated using default floating point arithmetic, its sign is guaranteed to be correct if its absolute value is greater than ε_β. Otherwise the filter fails.

Similarly, the semi-static filter for O^* is

$$\begin{aligned} \varepsilon_O &= 1.3865993466947057\,10^{-13}\delta_1\delta_2\delta_3\delta_4\delta_5\delta_6 \\ \delta_1 &= \max\{|r_x - s_x|, |u_x - t_x|, |v_x - t_x|, |r_x - t_x|\} \\ \delta_2 &= \max\{|r_y - s_y|, |u_y - t_y|, |v_y - t_y|, |r_y - t_y|\} \\ \delta_3 &= \max\{|r_z - s_z|, |u_z - t_z|, |v_z - t_z|, |r_z - t_z|\} \\ \delta_4 &= \max\{|r_x - s_x|, |b_x - c_x|, |a_x - c_x|, |r_x - c_x|\} \\ \delta_5 &= \max\{|r_y - s_y|, |b_y - c_y|, |a_y - c_y|, |r_y - c_y|\} \\ \delta_6 &= \max\{|r_z - s_z|, |b_z - c_z|, |a_z - c_z|, |r_z - c_z|\}. \end{aligned}$$

If both the filters succeed, we simply compose the sign of β and O^* to return the value of the predicate. If any of them fails, we re-evaluate the expression using interval arithmetic. To keep our code self-contained, we implemented our custom `interval number` type, but any interval type provided by existing libraries (e.g., Boost [Schling 2011] or CGAL [Hemmer et al. 2019]) can be used.

When using interval arithmetic, both β and O^* will be computed and represented as intervals. If any of them contains the zero, their sign is ambiguous (i.e., the filter fails) and we re-evaluate the expression using floating point expansions. Even in this case, we implemented our own custom expansion number type, but any existing implementations can be used (e.g., the `expansion_nt` type provided by Geogram [Lévy 2019]). Since precision is arbitrary, this last evaluation is error free and the sign is guaranteed to be correct.

4.2 orient3d_TPI

Let $\boldsymbol{v}_i$, $\boldsymbol{w}_i$, and $\boldsymbol{u}_i$, $i = 1, 2, 3$ be three triplets of non-collinear points that define three planes $\mathcal{P}_v, \mathcal{P}_w, \mathcal{P}_u$ respectively. As for the line case let $\boldsymbol{q}_i$, $i = 1, 2, 3$, be three points defining a reference plane. Let us assume that $\mathcal{P}_v, \mathcal{P}_w, \mathcal{P}_u$ intersect at a single point $\boldsymbol{p}$ then

$$\texttt{orient3D\_TPI}(\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3, \boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3, \boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3, \boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3)$$

is the sign of the volume of tetrahedron $(\boldsymbol{p}, \boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3)$.

Let

$$\begin{aligned} \boldsymbol{n}_v &= (\boldsymbol{v}_2 - \boldsymbol{v}_1) \times (\boldsymbol{v}_3 - \boldsymbol{v}_2) \\ \boldsymbol{n}_w &= (\boldsymbol{w}_2 - \boldsymbol{w}_1) \times (\boldsymbol{w}_3 - \boldsymbol{w}_2) \\ \boldsymbol{n}_u &= (\boldsymbol{u}_2 - \boldsymbol{u}_1) \times (\boldsymbol{u}_3 - \boldsymbol{u}_2) \end{aligned}$$

and

$$n_x = \begin{vmatrix} p_v & n_{vy} & n_{vz} \\ p_w & n_{wy} & n_{wz} \\ p_u & n_{uy} & n_{uz} \end{vmatrix}, \; n_y = \begin{vmatrix} n_{vx} & p_v & n_{vz} \\ n_{wx} & p_w & n_{wz} \\ n_{ux} & p_u & n_{uz} \end{vmatrix}, \; n_z = \begin{vmatrix} n_{vx} & n_{vy} & p_v \\ n_{wx} & n_{wy} & p_w \\ n_{ux} & n_{uy} & p_u \end{vmatrix}$$

with

$$p_v = \mathbf{n_v} \cdot \mathbf{v_1}, \; p_w = \mathbf{n_w} \cdot \mathbf{w_1}, \text{ and } p_u = \mathbf{n_u} \cdot \mathbf{u_1}.$$

Then

$$\boldsymbol{p} = \frac{1}{\beta}\boldsymbol{n} \quad \text{with} \quad \boldsymbol{n} = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix}^T \quad \text{and} \quad \beta = \begin{vmatrix} \boldsymbol{n}_v \\ \boldsymbol{n}_w \\ \boldsymbol{n}_u \end{vmatrix}.$$

If $\beta = 0$, $\boldsymbol{p}$ is undefined. Otherwise, `orient3d`$(\boldsymbol{p}, \boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3)$ is:

$$O(\boldsymbol{p}, \boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3) = \begin{vmatrix} \boldsymbol{p} - \boldsymbol{q}_3 \\ \boldsymbol{q}_1 - \boldsymbol{q}_3 \\ \boldsymbol{q}_2 - q_3 \end{vmatrix} = \frac{1}{\beta^3} \begin{vmatrix} \beta\boldsymbol{p} - \beta\boldsymbol{q}_3 \\ \beta\boldsymbol{q}_1 - \beta\boldsymbol{q}_3 \\ \beta\boldsymbol{q}_2 - \beta\boldsymbol{q}_3 \end{vmatrix} = \frac{1}{\beta^3} \begin{vmatrix} \boldsymbol{n} - \beta\boldsymbol{q}_3 \\ \beta\boldsymbol{q}_1 - \beta\boldsymbol{q}_3 \\ \beta\boldsymbol{q}_2 - \beta\boldsymbol{q}_3 \end{vmatrix}$$

As for the LPI case, this expression uses input values only and its sign can be obtained by composing the sign of β with the sign of

$$O^*(\boldsymbol{p}, \boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3) = \beta^3 O(\boldsymbol{p}, \boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3) = \begin{vmatrix} \boldsymbol{n} - \beta\boldsymbol{q} \\ \beta\boldsymbol{q}_1 - \beta\boldsymbol{q}_3 \\ \beta\boldsymbol{q}_2 - \beta\boldsymbol{q}_3 \end{vmatrix}$$

Again, the final answer follows from

$$\text{sign}(O) = \begin{cases} -\text{sign}(O^*) & \text{for } \beta < 0 \\ \text{sign}(O^*) & \text{otherwise} \end{cases}.$$

The semi-static filter, as calculated by [Meyer and Pion 2008], for the polynomial expression of β is

$$\begin{aligned} \varepsilon_\beta &= 8.8881169117764924\,10^{-14}\delta_1\delta_2\delta_3\delta_4\delta_5\delta_6 \\ \delta_1 &= \max\{|v_{2x} - v_{1x}|, |v_{3x} - v_{2x}|, |w_{2x} - w_{1x}|, |w_{3x} - w_{2x}|\} \\ \delta_2 &= \max\{|v_{2y} - v_{1y}|, |v_{3y} - v_{2y}|, |w_{2y} - w_{1y}|, |w_{3y} - w_{2y}|\} \\ \delta_3 &= \max\{|v_{2z} - v_{1z}|, |v_{3z} - v_{2z}|, |w_{2z} - w_{1z}|, |w_{3z} - w_{2z}|\} \\ \delta_4 &= \max\{|u_{2x} - u_{1x}|, |u_{3x} - u_{2x}|, |w_{2x} - w_{1x}|, |w_{3x} - w_{2x}|\} \\ \delta_5 &= \max\{|u_{2y} - u_{1y}|, |u_{3y} - u_{2y}|, |w_{2y} - w_{1y}|, |w_{3y} - w_{2y}|\} \\ \delta_6 &= \max\{|u_{2z} - u_{1z}|, |u_{3z} - u_{2z}|, |w_{2z} - w_{1z}|, |w_{3z} - w_{2z}|\} \end{aligned}$$

while the semi-static filter for the polynomial expression of O^* is

$$
\begin{aligned}
\varepsilon_O &= 3.4025182954957945\,10^{-12}\delta_1\delta_2\delta_3\delta_4\delta_5\delta_6\delta_7\delta_8\\
\delta_1 &= \max\{|v_{2x}-v_{1x}|, |v_{3x}-v_{2x}|, \lambda_x\}\\
\delta_2 &= \max\{|v_{2y}-v_{1y}|, |v_{3y}-v_{2y}|, \lambda_y\}\\
\delta_3 &= \max\{|v_{2z}-v_{1z}|, |v_{3z}-v_{2z}|, \lambda_z\}\\
\delta_4 &= \max\{\delta_1, \delta_2\}\\
\delta_5 &= \max\{\delta_2, \delta_3\}\\
\delta_6 &= \max\{\lambda_x, |q_{1x}-q_{3x}|, |q_{2x}-q_{3x}|\}\\
\delta_7 &= \max\{\lambda_x, \lambda_y, \lambda_z, |q_{1y}-q_{3y}|, |q_{2y}-q_{3y}|\}\\
\delta_8 &= \max\{|q_{1y}-q_{3y}|, |q_{2y}-q_{3y}|, |q_{1z}-q_{3z}|, |q_{2z}-q_{3z}|\}\\
\lambda_x &= \max\{|w_{2x}-w_{1x}|, |w_{3x}-w_{2x}|, |u_{2x}-u_{1x}|, |u_{3x}-u_{2x}|\}\\
\lambda_y &= \max\{|w_{2y}-w_{1y}|, |w_{3y}-w_{2y}|, |u_{2y}-u_{1y}|, |u_{3y}-u_{2y}|\}\\
\lambda_z &= \max\{|w_{2z}-w_{1z}|, |w_{3z}-w_{2z}|, |u_{2z}-u_{1z}|, |u_{3z}-u_{2z}|\}.
\end{aligned}
$$

We observe that Campen and Kobbelt [Campen and Kobbelt 2010a] use the filtered predicates proposed in [Bernstein and Fussell 2009] to exactly determine the position of a TPI point with respect to a reference plane. This is similar to what we do, but with a fundamental difference: in their approach, each plane is represented by the four coefficients of its implicit equation, while we use a vertex triplet. This requires a conversion, which in [Bernstein and Fussell 2009] is not exact and requires repairing, whereas in [Campen and Kobbelt 2010a] is made exact by first splitting long edges. The latter approach has a twofold disadvantage as it introduces new constructions (the splitting points must be calculated, with potential inaccuracy) and increases the size of the input (in particular for models with large edge length variation). In contrast, our `orient3d_TPI` operates directly on the input values, and it guarantees exactness without requiring any modification.

5 RESULTS

Our algorithm is implemented in C++ and uses Eigen [Guennebaud et al. 2010] for the linear algebra routines and Geogram [Lévy 2019] for the standard orientation predicates. We run our experiments on cluster nodes with a Xeon E5-2690v4 2.6GHz, limiting every job running time to 24 hours and maximum memory to 8GB. The reference implementation is open-source and available on GitHub: https://github.com/wangbolun300/fast-envelope.

5.1 Comparison with Inexact Methods

We perform a large-scale comparison of our method with two *inexact* methods: the sampling approach used in [Hu et al. 2020, 2018] and the Hausdorff bound (abbreviated as HB from now on) proposed in [Tang et al. 2009]. While the sampling approach is inherently approximate, the latter could be made exact by using rational numbers, but with an impractical running time. For this comparison we used a floating point reimplementation [Martin Skrodzki 2019].

Datasets. We run our algorithm on the Thingi10k dataset [Zhou and Jacobson 2016], which contains 10 000 real-world surface triangle meshes, and generated two sets of queries for each model. While our algorithm processed our benchmark on all models in less than 24 hours, there are 205 models where either HB or sampling run out of time. For fairness, we present all the statistics excluding them.

(1) **QSlim**. We modify the QSlim [Garland and Heckbert 1997] algorithm to stay within an envelope: we prevent any collapse that move the mesh outside the envelope and recorded all queries (Section 6.1).
(2) **TetWild.** We run the Tetwild [Hu et al. 2020, 2018] meshing algorithm and recorded the first 100k queries.

In both cases, the queries are used to validate the local operations inside the algorithm, to ensure that the output surface does not leave the envelope.

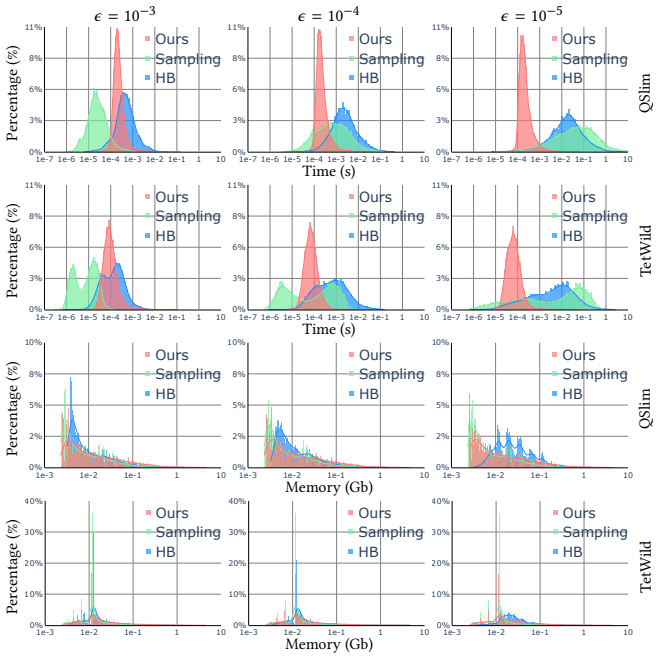


Fig. 13. Log histogram of the the average query time and peak memory of sampling, HB, and our method over the Thingi10k dataset for different envelope sizes and different dataset.

Envelope Size. For every method we use an envelope size ϵ proportional to the diagonal of the bounding box of the model and one that ensures (up to floating point arithmetic for the other two methods) that the query triangles are inside the L^2 envelope. Differently from using an explicit Minkowsky envelope (Figure 5), these three methods have a fast initialization (which is also amortized by the query time): Our average initialization time is 0.03 seconds, 0.004s for sampling, and 0.04s for HB, respectively. For large models our initialization can go up to 1.2s, compared to 3s and 0.2s for HB and sampling.

Large Dataset. In Figure 13 top, we compare the running time on both datasets for the three methods. For large envelopes, sampling is the fastest method, while both our algorithm and HB have similar performance. As the envelope shrinks, the performances of our method remains similar, while sampling and HB become slower.

In Figure 13, bottom shows the peak memory comparison. Our method has the overhead of storing the polyhedra and some local result to improve efficiency in addition to the search tree. Therefore our memory consumption is similar to the other two methods (maximum on the dataset of 1.44Gb versus 0.46Gb for sampling and

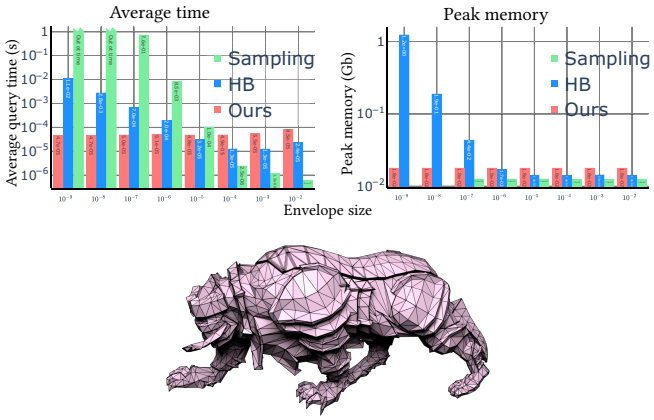


Fig. 14. Average query time and peak memory on one model (rendered) for increasingly smaller envelope for the sampling, HB, and our method, in log-log.

	HB	Ours	Ratio
max HB	2.58e-1	3.68e-3	70
max Ours	7.76e-2	2.39e-2	3
min HB	7.26e-6	7.65e-6	1
min Ours	1.06e-3	3.93e-6	269
avg	8.52e-3	4.56e-4	19
std	1.48e-2	6.01e-4	25

	HB	Ours	Ratio
max HB	2.20e-2	6.59e-4	33
max Ours	4.44e-3	2.50e-3	2
min HB	6.55e-6	7.92e-6	1
min Ours	1.29e-5	3.24e-6	4
avg	4.15e-4	1.90e-4	2
std	6.28e-4	1.44e-4	4

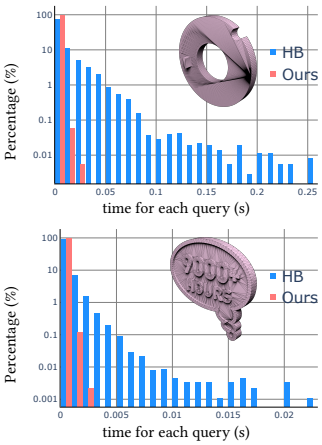


Fig. 15. Average over 100 runs of the query time for two similar models. The table shows the statistics for 30 000 queries. The first 4 rows shows the performances for the slowest/fastest query for the two different methods; for instance the first row shows the time for the slowest query for HB (2.58e-1) versus ours (3.68e-3). The histograms show the distribution of query time for the two methods. The $\#F$ of the top model is 3404, the $\#F$ of the bottom model is 3430.

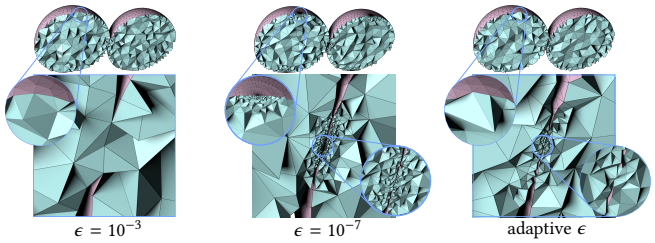


Fig. 16. Example of a model where the default envelope merges two input spheres, left. This can be fixed by globally shrinking the envelope (middle), or using an adaptive envelope, right.

1.91Gb for HB) and the trend is the same. As for the other method, the memory consumption depends only on the input mesh and not on the actual envelope size.

Small Envelope. Traditional envelope checks suffer in presence of thin envelopes. To measure this effect we consider a single model and vary the envelope size: in the experiment in Figure 14 we generated queries using TetWild with increasingly smaller envelope. Our method has similar running time independently from ϵ; the sampling method increases exponentially making it impossible to use for $\epsilon < 10^{-6}$; the HB method performs better than sampling but it also slows down for small envelopes.

We report the maximal memory usage for all methods in Figure 14. We observe that our method and sampling have similar requirements since they both store a spatial index of the input mesh, making the memory independent from the envelope size. In contrast HB requires more memory as the envelope shrinks since a smaller envelope leads to additional refinement steps.

Hausdorff. One of the limitations of the HB method is that the running time depends on the relative position of the query triangle and of the envelope. If its vertices are close to the envelope it would require several levels of refinement to sufficiently shrink the lower and upper bound. Figure 15 shows the average query time for different queries computed for two similar models. We see that the query time of our method is stable, leading to timings mostly independent from the query, while the HB method has large variation.

Summary. Overall, our approach has similar performances (in terms of both running time and memory consumption) as sampling and HB for large envelopes, while it is superior in the other cases. In addition to the performance benefits (and their invariance to envelope size), the exactness of our method is an important property in applications, as we will demonstrate in Section 6.

5.2 Adaptive Envelope

An additional feature of our approach, is that it is straightforward to have an envelope of varying size. This is a useful feature for controlling the deviation from the input surface during meshing. For instance, if two features are close, the remeshing algorithm could merge them (Figure 16, left) changing the number of component or topology of the output mesh. An easy solution would be to globally shrink the envelope, however this leads to higher running times and creates an unnecessarily fine surface everywhere (Figure 16, middle). Using an adaptive envelope one can have both: coarse mesh far from critical areas, and finer mesh where feature preservation is required (Figure 16, right), while keeping a reasonable running time. An adaptive envelope allows also to selectively preserve different features (Figure 17). On the top part of the rocket we can shrink the envelope to maintain the sharp features, while allowing the bottom part to be coarser.

5.3 Different Polyhedra

As mentioned in Section 3 our method requires only a set of convex polyhedra. For efficiency reasons we propose to use seven/eight

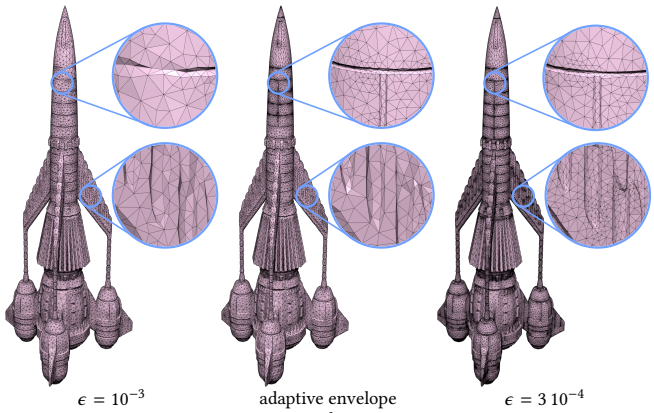


Fig. 17. Left: example of a model meshed using uniform $\epsilon = 10^{-3}$. Middle: using an adaptive envelope with $\epsilon = 3\,10^{-4}$ for the top part of the rocket to preserve the features, while on the bottom part uses $\epsilon = 10^{-3}$ to remove them. Right: meshing everything with $\epsilon = 3\,10^{-4}$ still preserves the features, but requires ~ 4x more faces.

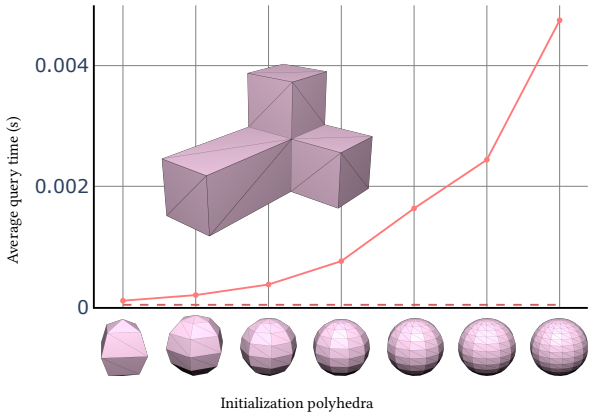


Fig. 18. Example of a simple mesh with different approximation of the true L^2 envelope. As we increase the number of facets of every polyhedron, the approximation becomes better at the expense of running time. The dashed line shows the query time using our polyhedral envelope.

planes (Section 3.1) with the downside of having a "rough" approximation of an Euclidean L^2 envelope. We can lift this limitation, at the expense of running time, by constructing a polyhedron per triangle which is closer and closer to the Euclidean envelope. In Figure 18 we use the Minkowsky sum between every triangle and differently dense approximations of a sphere. This produces, in the limit, an Euclidean envelope, however the cost per query increases as the polyhedra have more faces.

6 APPLICATIONS

We selected two testbed geometry processing applications to evaluate the performance of our algorithm in practical scenarios: surface remeshing (Section 6.1) and tetrahedralization (Section 6.2). We selected these two applications since they heavily rely on the envelope

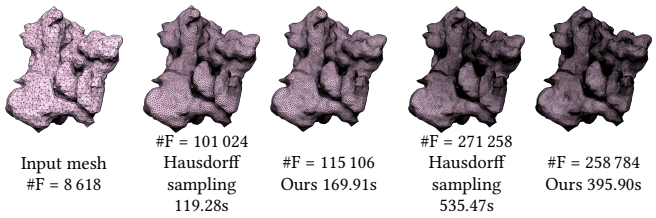


Fig. 19. Example of usage of our envelope in the method in [Cheng et al. 2019]. For two envelopes: 3e-3 (2nd and 3rd) and 2e-3 (4th and 5th).

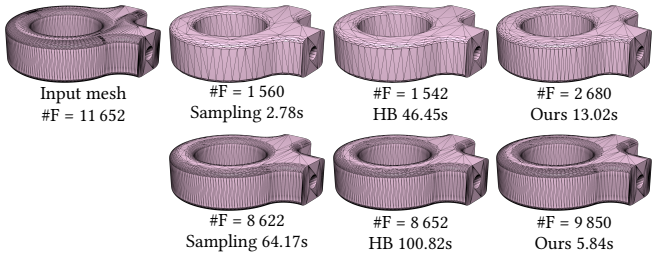


Fig. 20. Example of usage of our envelope in the method in [Garland and Heckbert 1997]. For two envelopes: 1e-3 (up) and 1e-4 (down).

check and provide open-source implementations, which allowed us to keep the original algorithm untouched and only replace the envelope containment check.

6.1 Surface Remeshing

We use two surface remeshing algorithms to test our envelope and compare it with alternatives: the remeshing method proposed in [Cheng et al. 2019] and a modified version of the QSlim [Garland and Heckbert 1997] algorithm.

[Cheng et al. 2019]. For the algorithm proposed in [Cheng et al. 2019] we use an envelope of 3e-3 and 2e-3 (Figure 19). The original implementation of [Cheng et al. 2019] uses a different sampling strategy based on Metro [Cignoni et al. 1998], which we replace with ours. For the larger envelope, our method has similar performance as Metro (which uses sampling) but provides guarantees that the final surface stays inside the envelope. When we shrink it, our method becomes faster since it avoids "locking" artefacts. We observed that the high running time for the sampling is caused by an additional remeshing step required to "push back" the mesh in the envelope. We note that both methods produce surfaces with similar quality.

QSlim. The QSlim algorithm is modified to guarantee an output within the envelope by: (1) rejecting operations moving triangles outside of the envelope, (2) always collapsing edges to one of their endpoints, and (3) stopping the algorithm when no valid edge collapse operation is left. We compare three ways of envelope containment check (Figure 20): the sampling method in [Hu et al. 2018], HB method in [Tang et al. 2009] and our envelope check, both using different envelope thickness of 10^{-3} and 10^{-4}. We note that, as visible in Figure 20, our method generates slightly denser meshes than HB or sampling since our envelope is thinner (due to the more conservative check).

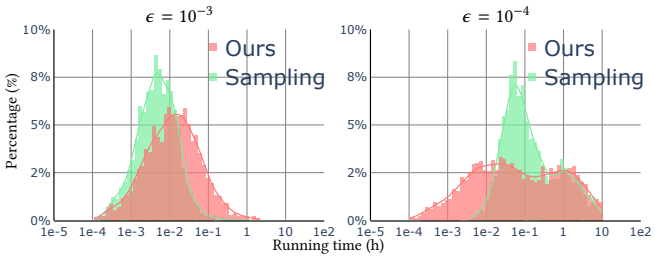


Fig. 21. Log plot of tetrahedralization of 2 000 models using our method compared with sampling for with two stages for different envelopes.

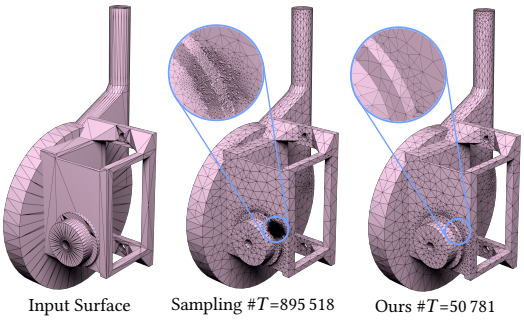


Fig. 22. Example a model (left) tetrahedralized where the inexact envelope (middle) triggers overrefinement (this effect can be removed using an inexact 3-stage envelope check). The problems completely disappears by using our envelope check.

6.2 Volumetric Meshing

We also integrated our envelope check in the fTetWild algorithm [Hu et al. 2020] and compared the running time on the dataset used in Section 5 (Figure 21). We note that fTetWild uses the multi-stage envelope check introduced in [Hu et al. 2018, Section 3.4] to compensate for the inexactness of the containment check. This procedure mitigates the locking effect but could still trigger unnecessary mesh refinements (Figure 22). Using our *exact* envelope check this heuristic becomes unnecessary (Figure 22) providing robustness and improving the output quality.

There is no algorithmic limitation in using small envelope sizes with fTetWild, which would be actually desirable to reproduce the input boundary with high geometric fidelity. However, the envelope check done with sampling becomes prohibitively expensive when the envelope thickness is thinner than 10^{-4}, making the algorithm impractical. Our approach solves this problem, having a running time mostly invariant to the envelope thickness (Figure 14). We show a series of tetrahedral meshes in Figure 23, using envelope of thickness varying from 10^{-2} to 10^{-8}. While the cost of the queries stays constant, the running time of the algorithm changes due to the different density of the result. The denser results obtained with a thinner envelope ensure that the final meshes preserve accurately the geometric details of the input.

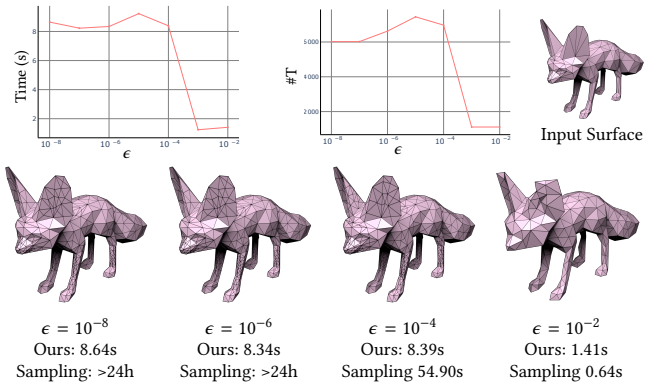


Fig. 23. Example of tetrahedral mesh for increasingly smaller envelope. The plots show the running time (left) and the number of tetrahedra (right) of our output meshes.

7 CONCLUDING REMARKS

We introduced a novel algorithm to exactly check for containment of a triangle inside the union of convex polyhedra. We integrated our algorithm in two remeshing applications and demonstrated that it avoids overrefinement, while at the same time allowing to use much smaller envelope sizes which lead to more geometrically accurate outputs.

While not important for remeshing applications, a limitation of our method is that it cannot directly check for containment in a L^2 envelope. While we can approximate the L^2 envelope with denser polyhedral approximations to increase accuracy, this negatively affects the runtime.

We believe that our contribution will become a useful tool in many geometry processing applications, and that our LPI and TPI predicates might find applications in other domains such as exact continuous collision detection. We plan to release an open-source reference implementation of both the predicates and our algorithm to foster adoption of our technique.

ACKNOWLEDGMENTS

This work was supported in part through the NYU IT High Performance Computing resources, services, and staff expertise. This work was partially supported by the NSF CAREER award under Grant No. 1652515, the NSF grants OAC-1835712, OIA-1937043, CHS-1908767, CHS-1901091, National Key Research and Development Program of China No. 2018YFB1107402, EU ERC Advanced Grant CHANGE No. 694515, a gift from Adobe Research, a gift from nTopology, and a gift from Advanced Micro Devices, Inc.

REFERENCES

Mikhail J. Atallah. 1983. A Linear Time Algorithm for the Hausdorff Distance Between Convex Polygons. *Inf. Process. Lett.* 17 (1983), 207–209.

Michael Barton, Iddo Hanniel, Gershon Elber, and Myung-Soo Kim. 2010. Precise Hausdorff distance computation between polygonal meshes. *Computer Aided Geometric Design* 27, 8 (2010), 580 – 591. Advances in Applied Geometry.

Gilbert Bernstein and Don Fussell. 2009. Fast, Exact, Linear Booleans. In *Proceedings of the Symposium on Geometry Processing (SGP âĂŹ09)*. Eurographics Association, Goslar, DEU, 1269âĂŞ1278.

H. Borouchaki and P. J. Frey. 2005. Simplification of surface mesh using Hausdorff envelope.

Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. 1998. Interval Arithmetic Yields Efficient Dynamic Filters for Computational Geometry. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry (SCG '98)*. ACM, New York, NY, USA, 165–174.
Marcel Campen and Leif Kobbelt. 2010a. Exact and Robust (Self-)Intersections for Polygonal Meshes. *Comput. Graph. Forum* 29 (05 2010), 397–406.
Marcel Campen and Leif Kobbelt. 2010b. Polygonal Boundary Evaluation of Minkowski Sums and Swept Volumes. *Computer Graphics Forum* 29, 5 (2010), 1613–1622.
Xiao-Xiang Cheng, Xiao-Ming Fu, Chi Zhang, and Shuangming Chai. 2019. Practical error-bounded remeshing by adaptive refinement. *Computers & Graphics* 82 (2019), 163 – 173.
Paolo Cignoni, Claudio Rocchini, and Roberto Scopigno. 1996. *Metro: Measuring Error on Simplified Surfaces.* Technical Report. Paris, France, France.
P. Cignoni, C. Rocchini, and R. Scopigno. 1998. Metro: measuring error on simplified surfaces. *Computer Graphics Forum* 17, 2 (1998), 167–174.
Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. 1996. Simplification Envelopes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH âĂŹ96)*. Association for Computing Machinery, New York, NY, USA, 119âĂŞ128.
Olivier Devillers and Sylvain Pion. 2003. Efficient exact geometric predicates for Delaunay triangulations. In *Procs. of 5th Workshop Algorithm Eng. Exper.* 37–44.
Steven Fortune and Christopher J. Van Wyk. 1996. Static Analysis Yields Efficient Exact Integer Arithmetic for Computational Geometry. *ACM Trans. Graph.* 15, 3 (July 1996), 223–248.
Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.* 33, 2, Article 13 (June 2007).
P. J. Frey and H. Borouchaki. 2003. Surface meshing using a geometric error estimate. *Internat. J. Numer. Methods Engrg.* 58, 2 (2003), 227–245.
Xiao-Ming Fu, Yang Liu, John Snyder, and Baining Guo. 2014. Anisotropic Simplicial Meshing Using Local Convex Functions. *IEEE Transactions on Visualization and Computer Graphics* (June 2014), 95–106.
Michael Garland and Paul S. Heckbert. 1997. Surface Simplification Using Quadric Error Metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH âĂŹ97)*. ACM Press/Addison-Wesley Publishing Co., USA, 209âĂŞ216.
Pijush K. Ghosh. 1993. A unified computational framework for Minkowski operations. *Computers & Graphics* 17, 4 (1993), 357 – 378.
Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3.
Peter Hachenberger. 2009. Exact Minkowksi Sums of Polyhedra andÂăExact andÂăEfficient Decomposition of Polyhedra into Convex Pieces. *Algorithmica* 55, 2 (01 Oct 2009), 329–345.
Michael Hemmer, Susan Hert, Sylvain Pion, and Stefan Schirra. 2019. Number Types. In *CGAL User and Reference Manual* (5.0 ed.). CGAL Editorial Board. https://doc.cgal.org/5.0/Manual/packages.html#PkgNumberTypes
Hugues Hoppe. 1996. Progressive Meshes. Association for Computing Machinery, Inc., 24.
K. Hu, D. Yan, D. Bommes, P. Alliez, and B. Benes. 2017. Error-Bounded and Feature Preserving Surface Remeshing with Minimal Angle Improvement. *IEEE Transactions on Visualization and Computer Graphics* 23, 12 (Dec 2017), 2560–2573.
Yixin Hu, Teseo Schneider, Xifeng Gao, Qingnan Zhou, Alec Jacobson, Denis Zorin, and Daniele Panozzo. 2019. TriWild: Robust Triangulation with Curve Constraints. *ACM Trans. Graph.* 38, 4, Article 52 (July 2019), 15 pages.
Yixin Hu, Teseo Schneider, Bolun Wang, Denis Zorin, and Daniele Panozzo. 2020. Fast Tetrahedral Meshing in the Wild. *ACM Trans. Graph.* 39, 4 (July 2020).
Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo. 2018. Tetrahedral Meshing in the Wild. *ACM Trans. Graph.* 37, 4, Article 60 (July 2018), 14 pages.
Mioara Joldes, Olivier Marty, Jean-Michel Muller, and Valentina Popescu. 2016. Arithmetic Algorithms for Extended Precision Using Floating-Point Expansions. *IEEE TRANSACTIONS ON COMPUTERS* 65, 4 (April 2016), 1197–1210.
Wonhyung Jung, Hayong Shin, and Byoung Kyu Choi. 2003. Self-intersection Removal in Triangular Mesh Offsetting.
Anil Kaul and Jarek Rossignac. 1992. Solid-interpolating deformations: Construction and animation of PIPs. *Computers & Graphics* 16, 1 (1992), 107 – 115.
Bruno Lévy. 2019. Geogram. http://alice.loria.fr/index.php/software/4-library/75-geogram.html.
C. Li, S. Pion, and C.K. Yap. 2005. Recent progress in exact geometric computation. *The Journal of Logic and Algebraic Programming* 64, 1 (2005), 85 – 111. Practical development of exact real number computation.
Manish Mandad, David Cohen-Steiner, and Pierre Alliez. 2015. Isotopic Approximation Within a Tolerance Volume. *ACM Trans. Graph.* 34, 4, Article 64 (July 2015), 12 pages.
S. Loriot Martin Skrodzki. 2019. https://github.com/martinskrodzki/cgal
Andreas Meyer and Sylvain Pion. 2008. FPG: A code generator for fast and certified geometric predicates. In *Real Numbers and Computers*. 47–60.
Sylvain Pion and Andreas Fabri. 2011. A generic lazy evaluation scheme for exact geometric computations. *Science of Computer Programming* 76, 4 (2011), 307 – 323. Special issue on library-centric software design (LCSD 2006).
Boris Schling. 2011. *The Boost C++ Libraries.* XML Press.
Jonathan Richard Shewchuk. 1997. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry* 18, 3 (1997), 305–363.
Hang Si and Jonathan Richard Shewchuk. 2014. Incrementally constructing and updating constrained Delaunay tetrahedralizations with finite-precision coordinates. *Engineering with Computers* 30, 2 (2014), 253–269.
Min Tang, Minkyoung Lee, and Young J. Kim. 2009. Interactive Hausdorff Distance Computation for General Polygonal Models. In *ACM SIGGRAPH 2009 Papers (SIGGRAPH '09)*. ACM, New York, NY, USA, Article 74, 9 pages.
Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10, 000 3D-Printing Models. *CoRR* abs/1605.04797 (2016). arXiv:1605.04797

A EXAMPLE OF REMESHING OPERATIONS LOCKED BY SAMPLING

During a remeshing process, both geometry and local connectivity may change to reach the objective of the algorithm. While doing this, keeping the triangle count as low as possible is a desirable feature. With reference to the 2D example in Figure 3: imagine that segment $[p, q]$ was introduced by previous remeshing steps while verifying that is contained within the envelope. This check was performed using the sampling approach: since its two vertices (black) and sample points (red) are all inside the envelope (blue), the segment was declared to be *inside*. At this point, the objective function of the remeshing algorithm requires that the segment $[p, q]$ is refined. To do so, the segment is split at its midpoint s. Since this operation does not change the geometry of $[p, q]$, the two resulting subsegments are assumed to be in the envelope without check. Now, imagine that the remeshing objective can be reached by slightly moving point q to a different position. Unfortunately, after such a move the check reveals that $[s, q]$ is outside the envelope because the yellow point is outside. Thus, the movement is prevented and replaced by a refinement. The segment is split again, another movement may not take place for the same reason, and another split occurs, and so on.

B EXAMPLE OF A NUMERICAL ISSUE IN LINE-PLANE INTERSECTION

Consider the following five points defined by their Cartesian coordinates:

$$\boldsymbol{a} = (0,0,0), \quad \boldsymbol{b} = (1,1,1),$$
$$\boldsymbol{r} = (1,0,0), \quad \boldsymbol{s} = (0,1,0), \quad \text{and} \quad \boldsymbol{t} = (0,0,1).$$

Let $\mathcal{L}$ be the straight line passing by $\boldsymbol{a}$ and $\boldsymbol{b}$, and let $\mathcal{P}$ be the plane spanned by $\boldsymbol{r}$, $\boldsymbol{s}$ and $\boldsymbol{t}$. The Cartesian coordinates of the intersection point $\boldsymbol{p} = \mathcal{L} \cap \mathcal{P}$ are $(1/3, 1/3, 1/3)$. The value $1/3$ has a repeating binary representation hence, when encoded as a floating point number, it must be necessarily truncated. Even if the truncation error is extremely small, the resulting value $0.333333\ldots$ is smaller than the actual value. That is why the predicate `orient3d`$(\boldsymbol{p}, \boldsymbol{r}, \boldsymbol{s}, \boldsymbol{t})$ returns 1 instead of 0. Conversely, our predicate `orient3d_LPI`$(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{r}, \boldsymbol{s}, \boldsymbol{t})$ returns zero, which is correct because the intersection point is exactly on the plane.