Exact and Efficient Intersection Resolution for Mesh Arrangements

JIA-PENG GUO and XIAO-MING FU*, University of Science and Technology of China, China

We propose a novel method to exactly and efficiently resolve intersections and self-intersections in triangle meshes. Our method contains two key components. First, we present a new concept of geometric predicates, called *indirect offset predicates*, to represent all intersection points through a new formulation and establish all necessary geometric predicates. Consequently, we reduce numerical errors in floating-point evaluations and improve the success rate of early stages of arithmetic filtering. Second, we develop localization and dimension reduction techniques for sorting, deduplicating, and locating the intersection points, thereby boosting efficiency and parallelism while maintaining accuracy. Rigorous testing confirms the robustness of our algorithm and consistency with previous methods. Comprehensive testing across diverse datasets further highlights the speed improvement achieved by our method, which is one order of magnitude faster than the state-of-the-art methods.

CCS Concepts: • **Computing methodologies** → **Shape modeling**.

Additional Key Words and Phrases: mesh arrangements, intersection resolution, constrained triangulation, geometric predicates

ACM Reference Format:
Jia-Peng Guo and Xiao-Ming Fu. 2024. Exact and Efficient Intersection Resolution for Mesh Arrangements. *ACM Trans. Graph.* 43, 6, Article 165 (December 2024), 14 pages. https://doi.org/10.1145/3687925

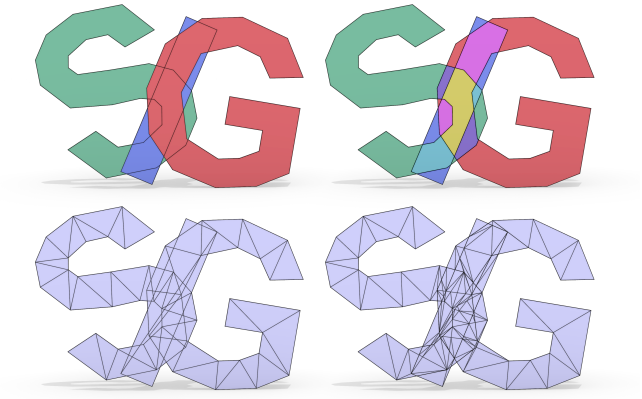


Fig. 1. Our algorithm exactly resolves intersections and self-intersections within general triangle meshes (left) without any additional assumption and produces triangulated results that completely preserve input geometry (bottom right). It enables mesh arrangements to further partition space into closed and consistently oriented cells (top right).

1 INTRODUCTION

In recent years, there has been a rapid growth in the quantity of 3D models, with one prominent category being triangle meshes [Koch et al. 2019; Zhou and Jacobson 2016]. While triangle meshes are extensively applied in modeling [Botsch et al. 2010, 2006; Botsch and Sorkine 2007; Sorkine 2006], simulation [Cheng et al. 2013; Si 2015], and manufacturing [Attene et al. 2018; Bermano et al. 2017; Livesu et al. 2017; Wang et al. 2013], the quality issues (e.g., intersection and self-intersection), are at odds with the strict requirements in many applications. Even if the input meshes are free from these issues, intersections are also introduced intentionally by specific applications, such as constructive solid geometry [Laidlaw et al. 1986]. It is essential to address and resolve these intersections to generate clean outputs.

Mesh arrangements convert meshes to a suitable form while preserving input geometry [Zhou et al. 2016]. It serves as the foundation for many high-level algorithms (e.g., mesh repair [Attene et al. 2013], boolean operations [Cherchi et al. 2022; Trettner et al. 2022], and volume meshing [Diazzi and Attene 2021; Hu et al. 2018]). We focus on the crucial initial stage of mesh arrangements: processing an arbitrary set of triangles without any additional assumptions about the input, and outputting an intersection-free simplicial complex that preserves the consistency of the original geometry (Figure 1).

To solve this problem, many algorithms and tools have adopted the well-established pipeline [Attene 2014; Attene et al. 2013; Cherchi et al. 2020; Zhou et al. 2016]: (1) intersection detection, where all pairs of triangles that mutually intersect are identified; (2) intersection classification, which determines the intersections (e.g., points and segments) between intersecting triangles; and (3) intersection elimination, which involves splitting triangles at their intersections to produce a valid configuration. While the robustness of these algorithms is guaranteed by various means, there remains significant room for improvement in efficiency. Generally, performance improvements are still limited by critical factors, such as the representation of intersection points, relevant geometric predicates (specifically, coordinate comparisons and point orientations), and associated sub-algorithms (encompassing sorting, deduplicating, and locating intersections).

Zhou et al. [2016] use rational numbers to exactly represent intersection points, yet leading to a non-negligible reduction in performance. Conversely, Cherchi et al. [2020] avoid explicitly constructing intersection points, instead implicitly representing them by combining geometric primitives, such as lines and planes, as suggested by [Attene 2020]. They reformulated the expression of these implicitly represented points as a fraction of polynomials and incorporated them into standard predicates using robust and efficient arithmetic filtering techniques [Shewchuk 1996, 1997] (for readers new to arithmetic filtering, see Paragraph "numerical methods" in Section 2). Evaluating these predicates is considerably faster than those relying on rational numbers, which is key to the success

*The corresponding author

Authors' address: Jia-Peng Guo, gjp171499@mail.ustc.edu.cn; Xiao-Ming Fu, fuxm@ustc.edu.cn, University of Science and Technology of China, China.

0730-0301/2024/12-ART165 $15.00
https://doi.org/10.1145/3687925

achieved by [Cherchi et al. 2020]. However, their polynomials include terms for both input point coordinates and their differences. In practice, intersections typically arise from nearby geometry primitives, indicating that the involved coordinate differences are often several orders of magnitude less than the coordinates themselves. Thus, the relatively larger coordinates terms in their polynomials lead to greater numerical errors in floating-point arithmetic, wider error bounds in (semi-)static filters [Lévy 2016; Meyer 2008], and, consequently, a higher failure rate of these filters.

To address the issues, we propose the concept of *indirect offset predicates*, which implicitly represent intersection points and reformulate their expression as a sum of base and offset parts. The base part is any input point for constructing the intersection point, while the offset part comprises polynomials that exclusively include terms for the differences between input coordinates. When the new expressions of intersection points are incorporated into standard predicates, the base part is subtracted by other base parts or input points, ensuring that the polynomials of indirect offset predicates maintain terms only for coordinates differences. Thus, by involving only the differences of input coordinates in floating-point calculations, we reduce the accumulated numerical error, narrow the error bounds, and enhance the success rate of filters.

We extend the framework of [Cherchi et al. 2020] to incorporate our new implicit points and predicates. However, their method relies on a global point set to deduplicate all intersection points and treats the intersection locating process as 3D problems solved by 3D intersection tests, which increases the computational burden and limits algorithm parallelism. To address these issues, we sort intersection points locally along 1D lines to eliminate duplicates and simplify the intersection locating process into a 2D problem. This greatly reduces the complexity and number of calls to predicates.

We test our algorithm on two datasets (one challenging dataset from [Zhou and Jacobson 2016] and one dataset created by us for stress testing purposes; see the construction details in Section 5) and compare our algorithm with the two latest algorithms [Cherchi et al. 2020; Zhou et al. 2016] that have the same problem settings as ours. Our algorithm is approximate 15 times faster than [Cherchi et al. 2020] and 56 times faster than [Zhou et al. 2016] on the Thingi10k dataset. On the stress-testing dataset, it is about 30 times faster than [Cherchi et al. 2020] and 75 times faster than [Zhou et al. 2016]. The results demonstrate the superior performance of our algorithm compared to both the comparison algorithms (Figure 2).

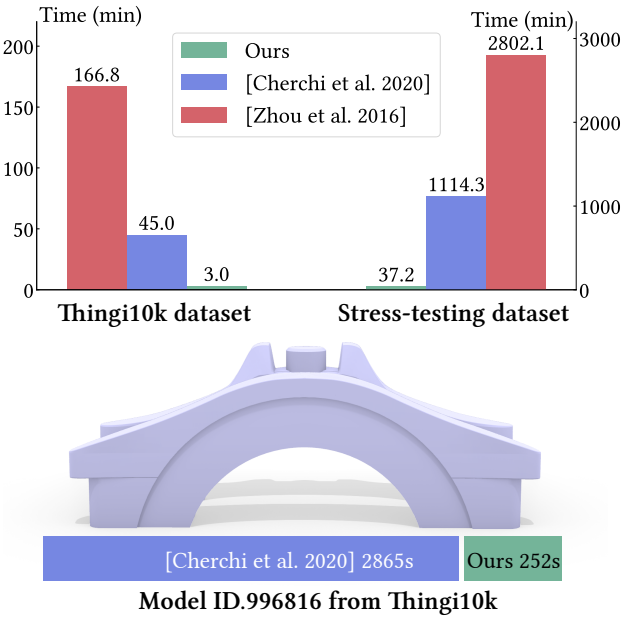


Fig. 2. We show the total runtime of ours and the other two algorithms [Cherchi et al. 2020; Zhou et al. 2016] on the Thingi10k dataset [Zhou and Jacobson 2016] and our constructed stress-testing dataset. We exclude the two most challenging models in the Thingi10K dataset (ID.996816 and ID.101633) from the previous statistics and show the former separately here (see statistics about the latter in Table 1).

2 RELATED WORK

Numerical methods. Geometric algorithms typically consist of two fundamental components, *predicates* and *constructors* [Fortune and Van Wyk 1996]. Arithmetic filtering techniques are developed to evaluate predicates exactly while minimizing performance loss [Devillers and Pion 2003; Shewchuk 1996, 1997]. It uses a series of floating-point filters (e.g., static filter [Fortune and Van Wyk 1993], semi-static filter [Meyer 2008], and dynamic filter [Brönnimann et al. 1998]) and switches to exact arithmetic if all filters fail. The (semi-)static filter estimates an upper bound of the rounding error (i.e., the error bound) for evaluating a predicate. The sign of the predicate is determined if the magnitude of its value evaluated in floating-point arithmetic exceeds the error bound. The static filter estimates a static error bound by analyzing the predicate expression based on given input value bounds. On the other hand, since the input value bounds and the error bound are positively correlated [Meyer 2008], the semi-static filter initially assumes input value bounds of 1 to evaluate the error bound, then scales it according to actual input value bounds.

However, many results generated by constructors still require high-precision numbers, hindering the use of these strategies. For example, the intersections are represented by rational numbers [Hemmer et al. 2023; Hu et al. 2018; Zhou et al. 2016] or fixed-precision rational numbers [Nehring-Wirxel et al. 2021; Trettner et al. 2022]. Attene [2020] represent intersection points implicitly through geometric constructions over input points, rewrite these points as fractions of polynomials, and reduce the predicates containing these points to fractions of polynomials. Therefore, arithmetic filtering can be seamlessly applied to evaluate predicates containing intersection points. To apply (semi-)static filters, implicit points' value bound and error bound are first estimated and then substituted into the predicate's expression for further error propagation. However, large input value bounds for their intersection points lead to wide error bounds. To address this, we propose *indirect offset predicates*.

Mesh arrangements. We focus on the crucial initial stage of mesh arrangements, i.e., intersection resolution. Two established methods (each associated with its corresponding geometry representation) exist for this problem: vertex-based [Cherchi et al. 2020; Zhou et al. 2016] and plane-based [Bernstein and Fussell 2009; Campen and

Kobbelt 2010; Trettner et al. 2022]. Plane-based methods are commonly seen in CSG applications, with EMBER [Trettner et al. 2022] being the fastest among all methods under specific assumptions in CSG scenarios. Vertex-based methods are widely adopted across various applications [Attene et al. 2013; Diazzi and Attene 2021; Diazzi et al. 2023; Hu et al. 2018; Zheng et al. 2024]. However, different methods always entail different objectives and requirements when targeted toward different applications, thus they cannot always be completely equivalent. The conversion between vertex-based and plane-based representations is also not trivial and is often accompanied by a loss of precision [Bernstein and Fussell 2009; Nehring-Wirxel et al. 2021; Trettner et al. 2022; Wang and Manocha 2013]. Considering that the exactness of the entire process can be guaranteed if algorithms and applications are implemented under the same geometric representations (e.g., the geometric kernels in CGAL [Brönnimann et al. 2024] and the indirect predicates [Attene 2020]), we develop our vertex-based algorithm under the same general problem settings as [Cherchi et al. 2020; Zhou et al. 2016] to meet the requirements of most applications.

Intersection detection. Detecting intersections among triangle meshes is a common task in geometric processing. To enhance the efficiency of this process, spatial partitioning is commonly employed, enabling intersection checks within each sub-space. Bounding Volume Hierarchy (BVH) is often favored due to its superior balance between runtime efficiency and memory usage. Zhou et al. [2016] and Barki et al. [2015] use a segment tree [Zomorodian and Edelsbrunner 2000], Cherchi et al. [2020] and Attene [2014] use a KDTree, and Cherchi et al. [2022] and Park [2004] use an OcTree with a common partition strategy (partitioning the nodes to eight sub-nodes with equal volumes). In CSG applications where triangles originate from different solids without self-intersections, more tailored strategies are developed to boost speed. Sheng et al. [2018] use an adaptive OcTree strategy that stops node partition once a node exclusively contains triangles from a single mesh. Trettner et al. [2022] use an AABB tree with a "median split" approach to separate a subset of a solid from the sub-nodes as early as possible. While these strategies are effective in these specific scenarios, their performance may diminish in others, such as with triangles from a single self-intersecting mesh. We adopt a general and effective adaptive partition strategy, which is detailed in Section 4.3.

Snap rounding. New degeneracy and self-intersections may be introduced again when naively rounding exact points to floating-point representation. This long and open problem, known as snap rounding (or vertex rounding) [Devillers et al. 2018], exists in mesh arrangements and similar algorithms that utilize high-precision points. While solutions to this problem do exist [Fortune 1999], it is impractical due to its complexity. Though downstream applications can adopt the same point form to circumvent this problem [Cherchi et al. 2022] or relax the input requirements to tolerate it [Diazzi and Attene 2021], such approaches are not always practical for all downstream applications. Both [Cherchi et al. 2020; Zhou et al. 2016] propose similar heuristics to address this problem, involving rounding coordinates from double to single precision and running their algorithms again. Typically, after at most two such steps, nearly all results are free of the snap rounding problem. Our algorithm follows a similar heuristic.

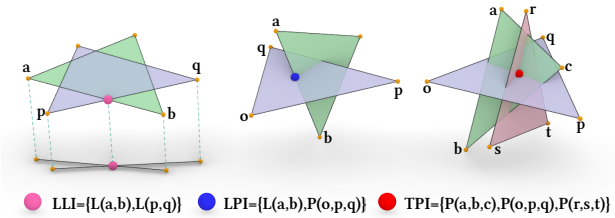


Fig. 3. Our algorithm defines three types of intersection points and corresponding implicit points, each constructed by lines and planes composed of explicit points (in yellow). Constructing LLI points requires assistance through projection onto orthogonal planes. We illustrate the LLI point constructed by two lines from coplanar triangles; they are also commonly constructed by lines from non-coplanar triangles.

3 OFFSET INDIRECT PREDICATES

Our new concept of predicates aligns with the core idea presented in [Attene 2020], which represents intersection points implicitly as combinations of input points and extends standard predicates to incorporate these *implicit points*, called *indirect predicates*. Our approach is founded on a different basis to avoid terms for large input coordinates (i.e., large input value bounds) in expressions of predicates. We introduce a new formulation for implicit points, consisting of base and offset parts. Based on the new formulation, the derivation of new predicate expressions contains terms exclusively for coordinate differences, achieving tighter error bounds and higher success rates in arithmetic filtering. These are referred to as *indirect offset predicates*. To evaluate the predicates, we sequentially apply semi-static filters, dynamic filters, and floating-point expansions [Lévy 2016] based on IEEE-754 double numbers.

3.1 Point Representation

We formulate the expression of an implicit point $\mathbf{i} = (i_x, i_y, i_z)$ as $\beta + \lambda/d = (\beta_x, \beta_y, \beta_z) + (\lambda_x, \lambda_y, \lambda_z)/d$, where the base part β is any input point (i.e., explicit point, abbreviated as EXP), and the offset part λ/d is a fraction of polynomials dedicated solely to the differences in input coordinates.

An implicit point becomes undefined ($d = 0$) in degenerate configurations of geometric primitives. Our algorithm ensures that the implicit point is well-defined by other checks (e.g., intersection detection) before construction. We need to determine the sign of d to determine the sign of predicates. A semi-static filter estimates a rounding error bound $\varepsilon_d = \varepsilon_0 \delta^n$ for d [Meyer 2008], where ε_0 and n are constants derived from the numerical error analysis of expression of d, and δ is the largest magnitude among the inputs. If $|d| > \varepsilon_d$, its sign is confirmed by the semi-static filter to be reliable. Otherwise, a dynamic filter evaluates the interval of d and confirms a reliable sign if its interval is unambiguous (i.e., does not overlap zero). If the dynamic filter fails, floating-point expansion exactly determines the sign.

We define three types of intersection points in our algorithm, each with its corresponding implicit point formulation (Figure 3). If

not explicitly distinguished, we use the same name interchangeably to refer to both intersection points and their corresponding implicit points.

3.1.1 Line-Line Intersection (LLI). We first define line-line intersection in 2D. Given two lines in 2D, each defined by two endpoints $(\mathbf{a}, \mathbf{b})$ and $(\mathbf{p}, \mathbf{q})$, their intersection point is formulated as below:

$$\mathbf{p}_S = (\beta_{Sx}, \beta_{Sy}) + (\lambda_{Sx}, \lambda_{Sy})/d_S,$$

where

$$\begin{aligned}
&\beta_{Sx} = a_x,\ \beta_{Sy} = a_y,\\
&\lambda_{Sx} = n_S(b_x - a_x),\ \lambda_{Sy} = n_S(b_y - a_y),\\
&d_S = \begin{vmatrix} a_x - b_x & a_y - b_y \\ q_x - p_x & q_y - p_y \end{vmatrix},\ n_S = \begin{vmatrix} a_x - p_x & a_y - p_y \\ q_x - p_x & q_y - p_y \end{vmatrix}.
\end{aligned}$$

Now we consider two lines $(\mathbf{a}, \mathbf{b})$ and $(\mathbf{p}, \mathbf{q})$ in 3D. We first find a proper orthogonal plane (XY, YZ, or ZX) to project two lines to avoid degeneration. Without loss of generality, we assume the found orthogonal plane is XY plane, and their intersection point is:

$$\begin{aligned}
\mathbf{p}_S &= (\beta_{Sx}, \beta_{Sy}, \beta_{Sz}) + (\lambda_{Sx}, \lambda_{Sy}, \lambda_{Sz})/d_S,\\
\beta_{Sz} &= a_z,\ \lambda_{Sz} = n_S(b_z - a_z),
\end{aligned}$$

where $\beta_{Sx}, \beta Sy, \lambda_{Sx}, \lambda_{Sy}, d_S, n_S$ are as defined in the above equation. The semi-static filter for d_S is:

$$\begin{aligned}
\varepsilon_{dS} &= 8.881784197001252\ 10^{-16}\delta_{dS}^2,\\
\delta_{dS} &= \max\{|a_x - b_x|, |a_y - b_y|, |q_x - p_x|, |q_y - p_y|\},\\
\delta_S &= \max\{\delta_{dS}, |a_x - p_x|, |a_y - p_y|\}.
\end{aligned}$$

Here, δ_{dS} is used in ε_{dS} and δ_S will be used in subsequent predicates' semi-static filters.

3.1.2 Line-Plane Intersection (LPI). Given a line and a plane in 3D, defined by two endpoints $(\mathbf{a}, \mathbf{b})$ and three points $(\mathbf{o}, \mathbf{p}, \mathbf{q})$ respectively, their intersection point is:

$$\mathbf{p}_L = (\beta_{Lx}, \beta_{Ly}, \beta_{Lz}) + (\lambda_{Lx}, \lambda_{Ly}, \lambda_{Lz})/d_L,$$

where

$$\begin{aligned}
&\beta_{Lx} = a_x,\ \beta_{Ly} = a_y,\ \beta_{Lz} = a_z,\\
&\lambda_{Lx} = n_L(b_x - a_x),\ \lambda_{Ly} = n_L(b_y - a_y),\ \lambda_{Lz} = n_L(b_z - a_z),\\
&d_L = \begin{vmatrix} \mathbf{a} - \mathbf{b} \\ \mathbf{p} - \mathbf{o} \\ \mathbf{q} - \mathbf{o} \end{vmatrix},\ n_L = \begin{vmatrix} \mathbf{a} - \mathbf{o} \\ \mathbf{p} - \mathbf{o} \\ \mathbf{q} - \mathbf{o} \end{vmatrix}.
\end{aligned}$$

The semi-static filter for d_L is:

$$\begin{aligned}
\varepsilon_{dL} =& 4.884981308350689\ 10^{-15}\delta_{dL}^3,\\
\delta_{dL} =& \max\{|a_x - b_x|, |a_y - b_y|, |a_z - b_z|, |p_x - o_x|,\\
&|p_y - o_y|, |p_z - o_z|, |q_x - o_x|, |q_y - o_y|, |q_z - o_z|\},\\
\delta_L =& \max\{\delta_{dL}, |a_x - o_x|, |a_y - o_y|, |a_z - o_z|\}.
\end{aligned}$$

3.1.3 Triplet-Plane Intersection (TPI). Given three planes in 3D, defined by three groups of points $(\mathbf{a}, \mathbf{b}, \mathbf{c})$, $(\mathbf{o}, \mathbf{p}, \mathbf{q})$ and $(\mathbf{r}, \mathbf{s}, \mathbf{t})$, their intersection point is:

$$\mathbf{p}_T = (\beta_{Tx}, \beta_{Ty}, \beta_{Tz}) + (\lambda_{Tx}, \lambda_{Ty}, \lambda_{Tz})/d_T,$$

where

$$\begin{aligned}
&\beta_{Tx} = a_x,\ \beta_{Ty} = a_y,\ \beta_{Tz} = a_z,\\
&d_T = \begin{vmatrix} \mathbf{n}_{opq} \cdot (\mathbf{b} - \mathbf{a}) & \mathbf{n}_{opq} \cdot (\mathbf{c} - \mathbf{a}) \\ \mathbf{n}_{rst} \cdot (\mathbf{b} - \mathbf{a}) & \mathbf{n}_{rst} \cdot (\mathbf{c} - \mathbf{a}) \end{vmatrix},\\
&\lambda_{Tx} = n_{Tu}(b_x - a_x) + n_{Tv}(c_x - a_x),\\
&\lambda_{Ty} = n_{Tu}(b_y - a_y) + n_{Tv}(c_y - a_y),\\
&\lambda_{Tz} = n_{Tu}(b_z - a_z) + n_{Tv}(c_z - a_z),\\
&n_{Tu} = \begin{vmatrix} \mathbf{n}_{opq} \cdot (\mathbf{o} - \mathbf{a}) & \mathbf{n}_{opq} \cdot (\mathbf{c} - \mathbf{a}) \\ \mathbf{n}_{rst} \cdot (\mathbf{r} - \mathbf{a}) & \mathbf{n}_{rst} \cdot (\mathbf{c} - \mathbf{a}) \end{vmatrix},\\
&n_{Tv} = \begin{vmatrix} \mathbf{n}_{opq} \cdot (\mathbf{b} - \mathbf{a}) & \mathbf{n}_{opq} \cdot (\mathbf{o} - \mathbf{a}) \\ \mathbf{n}_{rst} \cdot (\mathbf{b} - \mathbf{a}) & \mathbf{n}_{rst} \cdot (\mathbf{r} - \mathbf{a}) \end{vmatrix},\\
&\mathbf{n}_{opq} = (\mathbf{p} - \mathbf{o}) \times (\mathbf{q} - \mathbf{o}), \mathbf{n}_{rst} = (\mathbf{s} - \mathbf{r}) \times (\mathbf{t} - \mathbf{r}).
\end{aligned}$$

The semi-static filter for d_T is as follows:

$$\begin{aligned}
\varepsilon_{dT} =& 1.3145040611561864\ 10^{-13}\delta_{dT}^6,\\
\delta_{dT} =& \max\{|a_x - b_x|, |a_y - b_y|, |a_z - b_z|, |a_x - c_x|,\\
&|a_y - c_y|, |a_z - c_z|, |p_x - o_x|, |p_y - o_y|, |p_z - o_z|,\\
&|q_x - o_x|, |q_y - o_y|, |q_z - o_z|, |s_x - r_x|, |s_y - r_y|,\\
&|s_z - r_z|, |t_x - r_x|, |t_y - r_y|, |t_z - r_z|\},\\
\delta_T =& \max\{\delta_{dT}, |a_x - o_x|, |a_y - o_y|, |a_z - o_z|,\\
&|a_x - r_x|, |a_y - r_y|, |a_z - r_z|\}.
\end{aligned}$$

3.2 Point Comparison and Orientation

We replace the explicit points with this implicit expression in a standard predicate, converting the predicate's polynomial from Λ to a fraction of polynomials Δ/D. Assuming Λ involves only differences between explicit coordinates, Δ and D will similarly contain terms exclusively for coordinate differences. This is achieved by subtracting base parts from other base parts or explicit points during the conversion process. D is the product of d from multiple implicit points. The same process as determining the sign of d of implicit points is also applied to determine the sign of the numerator Δ.

In classifying and eliminating intersections, we sort intersection points along segments and locate them on planes. These tasks can be simplified into lower-dimensional operations. To this end, we introduce two predicates designed to determine the suitable dimension for projection robustly. Subsequently, we introduce two generalized predicates to perform orientation tests and coordinate comparisons on all combinations of implicit points in the reduced dimensions. For simplicity, we further abbreviate EXP, LLI, LPI, and TPI points as E, S, L, and T, respectively, to represent combinations of points accepted by predicates.

3.2.1 Robust Dimension Reduction. In our algorithm, intersection points are sorted along segments, with endpoints possibly being EXP, LLI, or LPI points, and are positioned on input triangles with only explicit vertices. We select dimensions for projection that ensure projected segments and triangles are far from degeneracy. This involves selecting the orthogonal axis (X, Y, Z) with the longest segment projection length and the orthogonal plane (XY, YZ, ZX) with the largest triangle projection area, respectively.

When the endpoints $\mathbf{a}$ and $\mathbf{b}$ of a segment are explicit points, the standard predicate `longestAxis`$(\mathbf{a}, \mathbf{b})$ identifies the axis where

the absolute difference in coordinates between the two points is maximized. When **a** and **b** are implicit points, arithmetic filtering is required to ensure that the magnitude of coordinate differences exceeds the numerical error. In this case, `longestAxis` evaluates the differences in coordinates between points across three axes, selects the axis with the greatest absolute difference and employs arithmetic filtering to prevent degeneration on that axis. For example, with **a** and **b** representing LPI and LLI points, the difference computation on the X axis is:

$$\frac{\Delta}{D} = \frac{(\beta_{Lx} - \beta_{Sx})d_L d_S + (\lambda_{Lx} d_S - \lambda_{Sx} d_L)}{d_L d_S}.$$

The semi-static filter for Δ is:

$$\varepsilon_\Delta = 6.084585960075558\ 10^{-14}\delta_\Delta^6,$$
$$\delta_\Delta = \max\{\delta_L, \delta_S, |\beta_{Lx} - \beta_{Sx}|\}.$$

The expressions and filters for the remaining axes and combinations of points are detailed in the supplementary material.

When projecting a triangle onto an orthogonal plane, ensuring that the projection area's magnitude exceeds the numerical error is necessary. The predicate `robustProject(a, b, c)` evaluates the projection area across three orthogonal planes and selects the plane with the largest area. Since all vertices **a**, **b**, **c** of a triangle are explicit points, we construct the semi-static filter for the projection area by the method proposed in [Meyer 2008] to prevent degenerate projection. We demonstrate this with an example on the YZ plane:

$$\Lambda = (c_y - b_y)(b_z - a_z) - (c_z - b_z)(b_y - a_y),$$
$$\varepsilon_\Lambda = 8.88720573725927976811\ 10^{-16}\delta_{\Lambda ab}\delta_{\Lambda bc},$$
$$\delta_{\Lambda ab} = \max\{|b_y - a_y|, |b_z - a_z|\},$$
$$\delta_{\Lambda bc} = \max\{|c_y - b_y|, |c_z - b_z|\}.$$

The expressions and filters for the remaining planes are similar; the only difference is the subscript for dimensions.

3.2.2 Generalized Comparison and Orientation. Coordinate comparison `pointCompare` and orientation tests on 2D `orient2d` are trivial for explicit points [Attene 2020], but not for implicit points. We demonstrate their generalization to the implicit points by showcasing one case for each. The remaining cases with different combinations of implicit points are presented in the supplementary material. Given an TPI point $\mathbf{p}_T$ and an LPI point $\mathbf{p}_L$ being compared on the X axis, the expression of `pointCompare_X_TL`$(\mathbf{p}_T, \mathbf{p}_L)$ is:

$$\frac{\Delta}{D} = \frac{(\beta_{Tx} - \beta_{Lx})d_T d_L + (\lambda_{Tx} d_L - \lambda_{Lx} d_T)}{d_T d_L}.$$

The signs of Δ, d_T, and d_L are evaluated to determine whether the expression's sign is negative, zero, or positive, which indicates whether the X-axis coordinate of $\mathbf{p}_T$ is less than, equal to, or greater than that of $\mathbf{p}_L$. The semi-static filter for Δ is:

$$\varepsilon_\Delta = 5.27253154330999\ 10^{-12}\delta_\Delta^{10},$$
$$\delta_\Delta = \max\{\delta_T, \delta_L, |\beta_{Tx} - \beta_{Lx}|\}.$$

When generalizing `orient2d` to test the orientation of three points $\mathbf{p}_T$, $\mathbf{p}_L$ and $\mathbf{p}_E$, which are TPI, LPI and EXP points respectively, on the YZ plane, the expression of `orient2d_YZ_TLE`$(\mathbf{p}_T, \mathbf{p}_L, \mathbf{p}_E)$ is $\Delta/D = \Delta/(d_T d_L)$, where Δ is:

$$\begin{aligned}\Delta =&(\lambda_{Ty} + d_T(\beta_{Ty} - p_{Ey}))(\lambda_{Lz} + d_L(\beta_{Lz} - p_{Ez}))\\ &- (\lambda_{Tz} + d_L(\beta_{Tz} - p_{Ez}))(\lambda_{Ly} + d_T(\beta_{Ly} - p_{Ey})).\end{aligned}$$

The signs of Δ, d_T, and d_L are evaluated to determine the sign of the orientation. The semi-static filter for Δ is:

$$\begin{aligned}\varepsilon_\Delta =&1.7707333863081813\ 10^{-11}\delta_\Delta^{11},\\ \delta_\Delta =&\max\{\delta_L, \delta_T, |\beta_{Ly} - p_{Ey}|, |\beta_{Lz} - p_{Ez}|,\\ &|\beta_{Ty} - p_{Ey}|, |\beta_{Tz} - p_{Ez}|\}.\end{aligned}$$

Compared to indirect predicates [Attene 2020; Cherchi et al. 2020], the semi-static filters for our predicates Δ typically have similar constants ε_0 and n in the ε_Δ. But our δ_Δ term only contains the differences between input coordinates. This distinction allows our δ_Δ^n to be significantly less than that in indirect predicates, making our predicates more likely to succeed in the semi-static filter phase and thus avoiding the need for expensive subsequent filters.

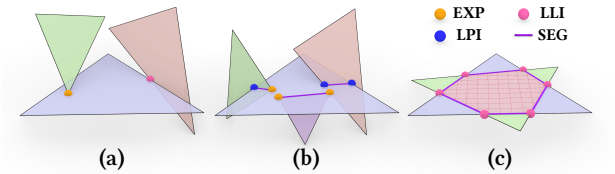


Fig. 4. The intersection between two intersecting triangles (except those forming a valid simplicial complex) may manifest as a point (a), a segment (b), or a convex polygon (c). The intersection points may be represented in different types of implicit points and are connected to form the intersection segments. We show only a subset of all possible combinations here.

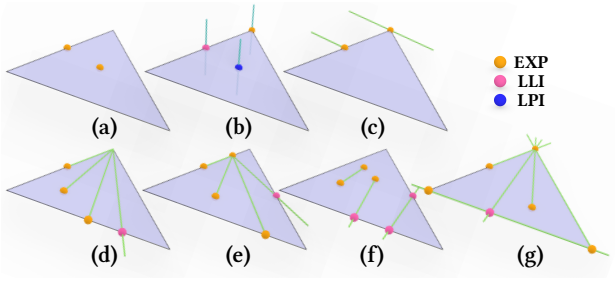


Fig. 5. Enumeration of all intersections between sub-simplices (vertices and edges) of one triangle and another triangle. They can be classified into three categories: (a) coplanar vertices, (b) crossing edges (in cyan), and (c~g) coplanar edges (in green). We display the intersection points for all of them.

4 INTERSECTION RESOLUTION

Algorithm overview. Our algorithm builds upon the foundation laid by [Cherchi et al. 2020, 2022], using our innovative predicates and integrating key optimization techniques to enhance the parallelism and speed up the algorithm. The algorithm can be conceptually divided into three steps: intersection detection, classification, and elimination. In intersection detection, possibly intersecting triangles are quickly found by an adaptive tree and confirmed by a triangle-triangle intersection detection routine (Section 4.1). Following this, intersection classification is performed for each pair

of intersection triangles to identify the location and type of intersections (Section 4.1). Once all intersections are identified, the elimination step applies a constrained triangulation to all triangles by treating intersections as constraints, thereby forming a valid simplicial complex (Section 4.2). Details about the adaptive tree and other implementation details are presented in Section 4.3. Additionally, our algorithm also includes some common preprocessing routines, such as removing duplicate vertices and triangles and deleting degenerate triangles, which we do not discuss further.

4.1 Intersection detection and classification

The triangle-triangle intersection (TTI) detection routine involves performing orientation tests of two triangles to uncover all possible intersections. These intersections may include points (which could be of types such as EXP, LLI, LPI, but not TPI), segments (whose endpoints are defined by intersection points), and polygons (formed by the segments) (see examples in Figure 4). Following this, the TTI classification routine identifies the location and types of the intersections by testing sub-simplices of one triangle against those of the other (Figure 5). Finally, intersection points and segments are attached to the corresponding triangles for subsequent triangulation.

4.1.1 Simplify intersection by LLI. Two edges from different triangles (whether these triangles are coplanar or not) will intersect to generate line-line intersection points ((b) and (d~g) in Figure 5), which are commonly detected in real-world models. For example, in the Thingi10k dataset, LLI points comprise approximately 21.4% of all generated implicit points, with LPI and TPI points constituting about 57.4% and 21.2%, respectively. Hence, we construct LLI implicit points to represent line-line intersection points rather than constructing LPI implicit points with an auxiliary point [Cherchi et al. 2020]. This reduces the computational burden and improves the success rate of filters for subsequent predicates.

4.1.2 Locally deduplicate intersection points. Each newly generated intersection point is assigned a unique index, which is used to reference it elsewhere. However, when multiple edges and triangles intersect at the same point, duplicate intersection points (those with identical geometry but different indices) are often generated. Removing these duplicates is crucial to prevent disjointed connectivity in the results.

In the classification step, only LLI and LPI points are constructed and may coincide with existing EXP, LLI, and LPI points. We store these points on the edges and sort them along the longest axis of each edge instead of sorting all points lexicographically within a global set [Cherchi et al. 2020]. In particular, to prevent race conditions (which occur when multiple threads access shared resources simultaneously in a parallel execution environment), it is necessary to lock an edge when operating on it. This edge-oriented localization has three main benefits:

(1) Sorting intersection points along edges breaks down the global sorting problem into many smaller local problems, reducing the scale of data for each sorting-related operation by several orders of magnitude (Figure 6).

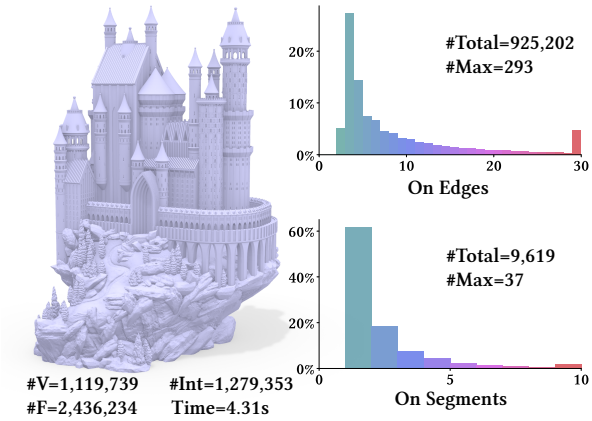


Fig. 6. We show the scale of intersection points, sorted along edges and intersection segments, for one of the most complex models (ID.1368052) in the Thingi10K dataset. We use histograms to show the distribution of the number of intersection points on edges and intersection segments. "#Total" is the total number of intersection points, and "#Max" is the maximum number of intersection points on any edge or segment. The number of vertices (#V), number of faces (#F), number of intersecting triangle pairs (#Int), and the runtime of our algorithm (Time) are also shown in the figure.

(2) By sorting points along the longest axis of edges, we limit comparisons to a single axis and avoid comparisons on a possible degenerate axis.
(3) Although each edge must be locked to avoid race conditions, new intersection points are typically generated on different edges simultaneously in a parallel execution environment. This means locking the edge does not significantly hinder thread performance, thereby facilitating the algorithm's parallel execution.

All these factors together substantially enhance the efficiency.

Sorting points facilitates a fast search to determine if identical points exist, thereby avoiding duplicate intersection points stored on the edge. When detecting a new intersection point, we *check* for duplicates to decide whether to *construct* a new point or *merge* duplicates. Specifically, when detecting an LPI point at the intersection of one edge and one triangle, if there is no geometrically identical point on that edge, a new implicit point is constructed and added to the edge; otherwise, the existing point is used to avoid duplicates. Similarly, when detecting an LLI point at the intersection of two edges, we check if geometrically identical points exist on the edges:

(1) If both edges share the same point (identical in geometry and index), no construction or merge is required.
(2) If both edges have geometrically identical points with different indices, these points are merged into the one with the simpler type and smaller index. Among types, an EXP point is simpler than an LLI point, which in turn is simpler than an LPI point; the simpler type has priority over the smaller index.
(3) If only one edge contains a geometrically identical point, it is then shared with the edge where it was absent.
(4) If neither edge has a geometrically coincident point, a new implicit point is constructed and shared between both edges.

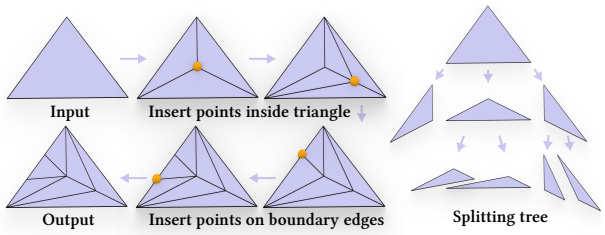
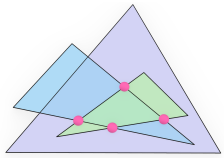


Fig. 7. Splitting a triangle by intersection points. The intersection points within the triangle are first inserted (upper row), and the sorted intersection points along the edges are sequentially inserted (bottom row). When the inner intersection points are too much, the newly generated sub-triangles are organized in a splitting tree (right), which is used to quickly locate points to insert.

Ultimately, only one implicit point will survive and be used for subsequent operations. Since each pair of edges intersecting at the same point undergoes the above check-construct-merge process for LLI points, only one point with the simplest type and smallest index is retained across all edges, ensuring simplicity and uniqueness.

4.1.3 Efficiently propagate coplanar intersections. Some intersection points between two coplanar triangles may lie within another triangle. For example, in the inset, the intersection points (shaded in red) between the blue and green triangles lie inside the purple triangle. However, since TTI detection and classification are conducted individually on pairs of triangles (e.g., blue-green, blue-purple, and green-purple), these red points are not recorded as constrained points for the purple triangles in the subsequent constrained triangulation. To address it, for each triangle, we propagate the intersection points from its coplanar edges to the triangle. Since points are already sorted along edges and the overlapping portion between edges and triangles has already been detected, we can efficiently traverse and propagate points within the overlapping portion without any geometric checks.



4.2 Intersection elimination

In this step, constrained triangulation is applied to each triangle by treating intersection points and segments as constraints. A linearized constrained 2D triangulation method [Livesu et al. 2022] is extended to incorporate indirect predicates [Cherchi et al. 2022]. It first inserts intersection points to split triangles (Figure 7), then inserts intersection segments as constrained segments following the general process [Shewchuk and Brown 2015]: locating, removing, and re-triangulating (Figure 8 (a)). During the locating phase, two segments may intersect, meaning that three triangles intersect, creating a new TPI point as a constrained point. We extend the entire process to incorporate our new implicit points and predicates and simplify critical parts.

4.2.1 Simplify intersection locating process. Triangulating a triangle t is fundamentally a 2D problem. However, extending the locating process to incorporate the proposed predicates is not trivial. Instead of relying on expensive and redundant 3D intersection tests to locate intersections in 3D space [Cherchi et al. 2020], we utilize the `robustProject` and the generalized `orient2d` to reduce the locating process to a lower dimensional one. Unfortunately, the generalized `orient2d` remains costly since it is applied to implicit points. To this end, we carefully simplify the locating process and reuse calculated results across the whole process, reducing the calls to the generalized `orient2d` as much as possible.

Specifically, we apply `robustProject` to t to find an orthogonal 2D plane where the projection of t is not degenerate, denoted as $\mathcal{P}_t$. Then, we compute the orientation O_t of t on the $\mathcal{P}_t$ by the standard `orient2d`$(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2)$ predicate, where $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$ are three oriented vertices of t and are projected to $\mathcal{P}_t$. O_t may be either positive or negative. By utilizing $\mathcal{P}_t$ and O_t, we locate a point in a splitting tree (Figure 9) and pass the calculated orientation across nodes, thus

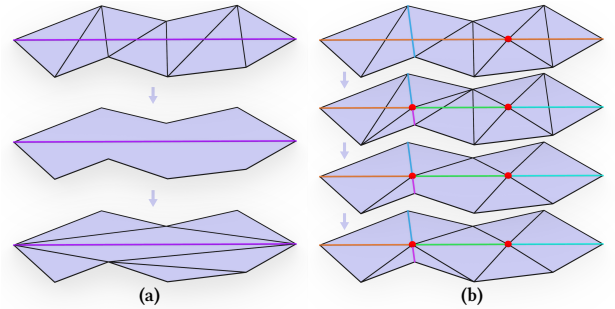


Fig. 8. Inserting a segment into a triangulation. (a) A segment (shaded in purple) is inserted into a triangulation (top), the intersecting triangles are removed (middle), and the void is re-triangulated by a linearized earcut method [Livesu et al. 2022] (bottom). (b) A horizontal segment (shaded in orange) is inserted into a triangulation, intersecting an existing constrained segment (shaded in blue) and an existing TPI point (shaded in red). A new TPI point is generated by intersecting two segments, splitting segments into five segments (shaded in different colors) together with the existing TPI point. Three horizontal segments are re-inserted, similar to (a).

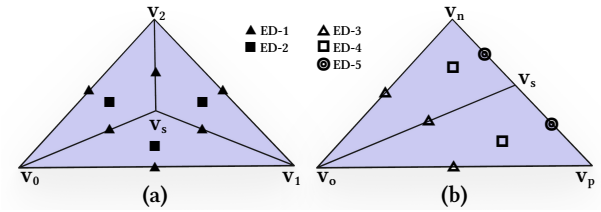


Fig. 9. An illustration of locating a point $\mathbf{p}$ in a node of a splitting tree. (a) In the case of a node with three sub-nodes, by evaluating generalized `orient2d`$(\mathbf{v}_s, \mathbf{p}, \mathbf{v}_j)$ on $\mathcal{P}_t$, where $j = 0, 1, 2$, the point is located in three possible positions, corresponding to end condition (ED) 1 to 2. (b) Similarly, for a node with two sub-nodes, by evaluating generalized `orient2d`$(\mathbf{v}_s, \mathbf{v}_o, \mathbf{p})$ on $\mathcal{P}_t$, the point is located in four possible positions, corresponding to ED-3 to ED-5. Each end condition triggers an iterative location query in the corresponding child node until the node is a leaf node. Specifically, when $\mathbf{p}$ is located on an edge in a node (corresponding to ED-1, ED-3, and ED-5), it must also be located on an edge in the sub-nodes. In this case, we pass the edge information to the subsequent sub-nodes to assist in locating $\mathbf{p}$. The location of $\mathbf{p}$ may conclude with end conditions ED-1 and ED-3 by solely topological checks. However, for the remaining case ED-5, a generalized `orient2d`$(\mathbf{v}_s, \mathbf{v}_o, \mathbf{p})$ is still required.

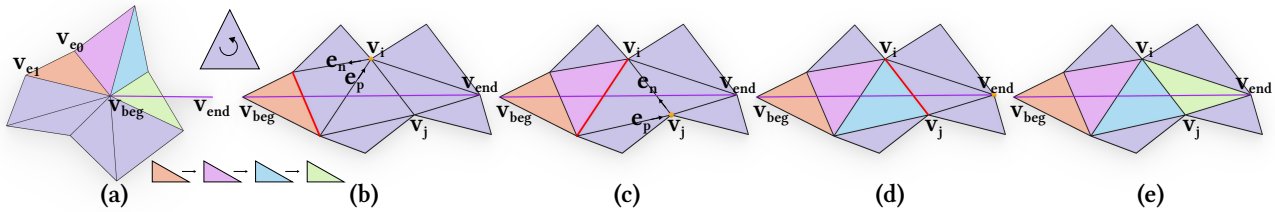


Fig. 10. Locating a segment $g = (\mathbf{v}_{\text{beg}}, \mathbf{v}_{\text{end}})$ in a triangulation. Without loss of generality, assuming O_t is positive in this illustration, thus, the triangle vertices are oriented in a counter-clockwise (CCW) direction. When determining the rotation direction in the initial step (a), we calculate orientation O_{e_0} and O_{e_1} for the points $(\mathbf{v}_{\text{beg}}, \mathbf{v}_{\text{end}}, \mathbf{v}_{e_0})$ and $(\mathbf{v}_{\text{beg}}, \mathbf{v}_{\text{end}}, \mathbf{v}_{e_1})$, respectively. If both O_{e_0} and O_{e_1} are the same as O_t (or neither matches O_t), we rotate clockwisely (or counter-clockwisely). This applies to orange, purple, and blue triangles. The rotation process stops when $O_{e_0} < 0$ and $O_{e_1} > 0$, thus, it stops at the green triangles. In the iterative step ((b)~(e)), we calculate the orientation O_{oppo} for points $((\mathbf{v}_{\text{beg}}, \mathbf{v}_{\text{end}}, \mathbf{v}_{\text{oppo}}))$, where $\mathbf{v}_{\text{oppo}}$ is the vertex (shaded in yellow) opposite to the gate edge (shaded in red), such as $\mathbf{v}_i$ in (b), $\mathbf{v}_j$ in (c), and $\mathbf{v}_{\text{end}}$ in (d). When O_{oppo} is positive (or negative), the next gate edge is the previous edge e_p (or the next edge e_n) of $\mathbf{v}_{\text{oppo}}$. The iterative process stops when $\mathbf{v}_{\text{oppo}}$ is the $\mathbf{v}_{\text{end}}$, $O_{\text{oppo}} = 0$, or the gate edge is a constrained edge.

limiting the calls to the generalized `orient2d` to a maximum of one or three times in a node with 2 or 3 sub-nodes.

Locating a segment includes an initial step to locate the first triangle intersecting with the segment and subsequent iterative steps to locate the remaining triangles. This process is illustrated and detailed in Figure 10. In the initial step, we reduce the expected number of calls to generalized `orient2d` to fewer than half the minimal vertex valence of the segment's end vertices. In the iterative step, only one call to generalized `orient2d` is required to identify each intersecting triangle.

4.2.2 Locally deduplicate TPI points. When inserting a segment, it may intersect with a previously inserted segment. These two segments must span at least three non-coplanar triangles, forming an unidentified TPI point. Similarly, it might also intersect with an existing TPI point. In both scenarios, it is split at the TPI point, and the resulting sub-segments are then re-inserted (Figure 8 (b)). Since the triangles are triangulated individually, duplicate TPI points are often unavoidably created on different triangles and must be deduplicated.

A TPI point is created by intersecting two intersection segments, which is similar to creating an LLI point by intersecting two triangle edges. Thus, we store TPI points on the intersection segments along the longest axis of each intersection segment. The longest axis is selected by applying `longestAxis` to the segment. Then, we follow the similar check-construct-merge process to the LLI points introduced in Section 4.1.2 to deduplicate TPI points. Locally sorting and deduplicating TPI points also benefit from the advantages mentioned in Section 4.1.2 and showcased in Figure 6. Moreover, this sorting does not involve input points or previously detected LLI and LPI points. All of these make sorting and deduplicating no longer a bottleneck in this step.

4.2.3 Triangulation and synchronization. After locating and possibly splitting an intersection segment, the intersecting triangles are removed, leaving two void polygons for triangulation. For details and terminology on the polygon triangulation method, see [Livesu et al. 2022]. Here, we briefly introduce the method and a new property. The polygons may contain dangling edges, holes, or both, complicating the process. Livesu et al. [2022] address this by extracting *simple polygons* using topological duplication of vertices or edges. They prove that a simple polygon always contains an *internal ear*, allowing for linear complexity triangulation by progressively cutting these ears. We further introduce a new property: a TPI vertex (except the extrema) is always convex and forms an internal ear, as it results from the internal intersection of two constrained segments, with its internal angle always less than π. Thus, we can cut all TPI vertices without geometric checks and the remaining vertices as before. The property reduces the need for most generalized `orient2d` checks on TPI points, which are the relatively expensive predicates in our algorithm.

Finally, since all triangles are triangulated individually, we apply the synchronization method in [Cherchi et al. 2020] to ensure consistent triangulation of overlapping polygon pockets between coplanar triangles. This method establishes a global mapping from the vertex indices of coplanar pockets to their triangulation, ensuring unique and consistent triangulation between overlapping pockets with identical vertex indices.

4.3 Implementation details

Adaptive tree. When building a BVH, three key considerations are partitioning, splitting, and stopping criteria. Conventional methods partition the node into a fixed number of children by splitting at the node's center, stopping when the node contains fewer than a specified threshold of triangles [Cherchi et al. 2022; Park 2004; Sheng et al. 2018]. In specific applications, heuristic methods are proposed to select split points to simplify subproblems [Trettner et al. 2022]. In our case, it is crucial to avoid assigning triangles to multiple nodes to prevent redundant intersection tests. To address this, We introduce an adaptive strategy that considers both the partitioning and stopping criteria.

Our strategy works as follows: given a split point, if the portion of triangles assigned to both sides of the split point along one axis is less than a specified *adaptive threshold* (e.g., 0.2), we partition the node along that axis. Depending on the number of axes along which the partitions occur, the node may be partitioned into eight, four, or two sub-nodes. Partitioning stops when partition does not occur on any axis or when the number of triangles in the node drops below a specified *split threshold* (e.g., 400). While finding an optimal split point to minimize triangles in leaf nodes is attractive, splitting at the node's center is already effective in balancing construction time and intersection detection time.

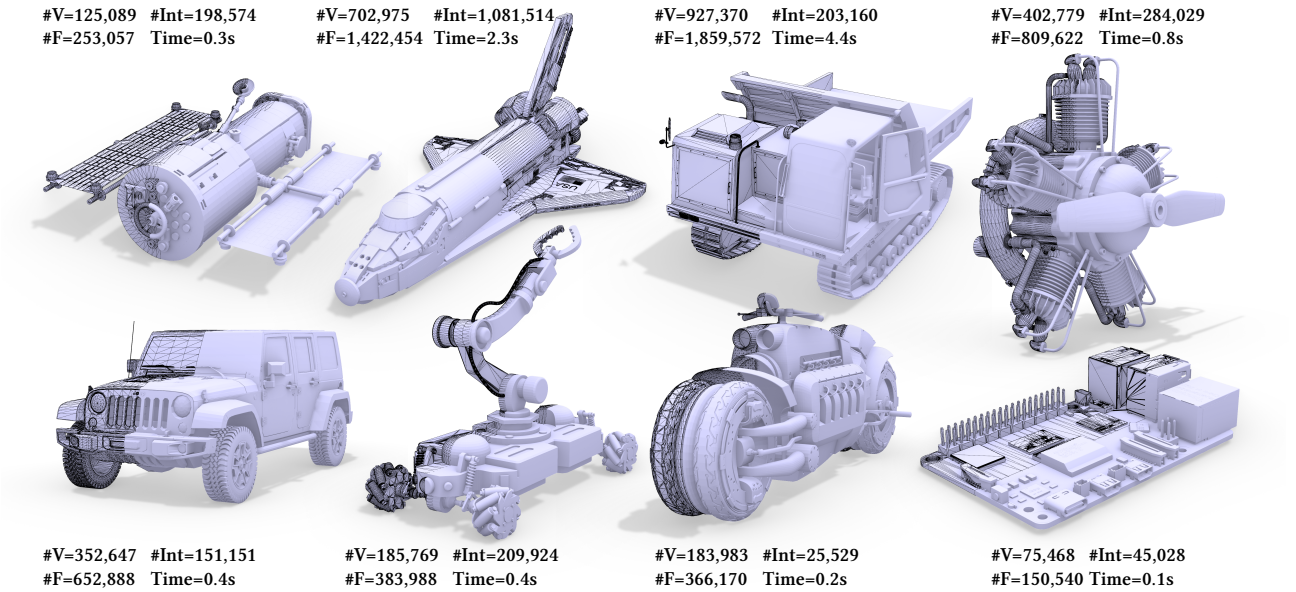


Fig. 11. Industrial CAD models from GrabCAD [Stratasys 2024], containing numerous self-intersections. The intersections are resolved by our algorithm.

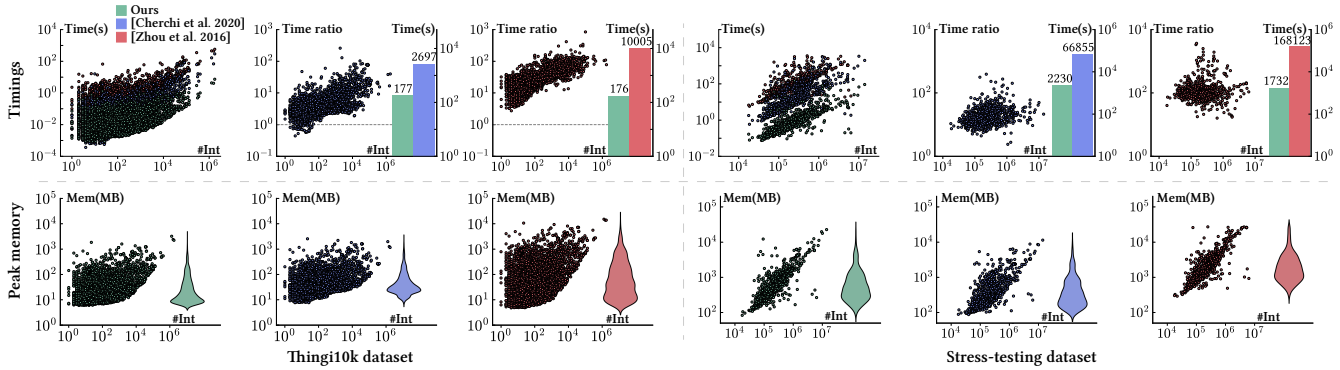


Fig. 12. Comparisons of timings and peak memory usage across our algorithm (green) and two competitors, [Cherchi et al. 2020] (blue) and [Zhou et al. 2016] (red), evaluated on two datasets. The upper row shows the runtime distribution (Time vs. #Int) and the distribution of relative performance (Time ratio vs. #Int, calculated as [Cherchi et al. 2020]/Ours and [Zhou et al. 2016]/Ours, respectively) by scatter plots. The overall runtime comparisons across three algorithms are shown by bar charts, with corresponding runtime values annotated above each bar. The bottom row displays the peak memory distribution (Mem vs. #Int) by scatter and violin plots. For clarity, all axes are displayed on a logarithmic scale.

Switch between exact arithmetics. In floating-point expansions [Joldes et al. 2016], an exact number is represented as the sum of multiple IEEE-754 64-bit floating-point numbers. The complexity of multiplying two exact numbers in expansions is $O(mn)$, where m and n are counts of floating-point numbers composing the exact numbers (also known as the length of expansions). Typically, implicit point coordinates can be exactly represented by summing a few floating-point numbers (mostly less than 6 in the Thingi10k dataset), making multiplication in expansions faster than that in rational arithmetic. However, multiplication in expansions may become slower than that in rational arithmetic when m and n become large. Therefore, once we detect that the multiplication may be too complex during evaluating predicates (e.g., mn >100), we switch to rational arithmetic to continue evaluation. This heuristic strategy has no adverse effects on simple models (it does not increase the overall runtime of the Thingi10k dataset) but significantly reduces the runtime for extremely complex models. For example, the runtime of model ID.101633 with and without this strategy is 310 seconds and 1186 seconds, respectively. The TPI point coordinates of the model ID.101633 often exceed a length of 10 in floating-point expansions.

Local cache. We cache the implicit expression of our implicit points. Our algorithm exhibits good locality in two aspects: (1) we sort and deduplicate implicit points on edges and segments separately; (2) we triangulate triangles individually. Consequently, we only cache the expressions within a restricted region and clear the cache before entering the next one. This localized caching strategy notably enhances efficiency without causing unacceptable memory usage (Figure 12).

5 EXPERIMENTS

We implement our algorithm in C++ and conduct all experiments on a PC with an Intel i5-13400F processor (2.5 GHz, 10 physical cores, and 16 logical cores) and 32 GB RAM. Our implementation relies on the parallel framework and concurrent data structures (e.g., concurrent vector) from Intel's TBB. Since competitors provide parallel implementations and our main contribution lies in parallelization, we use our parallel implementation for most experiments and comparisons. For some models and results displayed in the figures, we report the number of vertices (#V), number of faces (#F), and number of intersecting triangle pairs (#Int). #Int is also used to indicate the model's complexity in our experiments. To facilitate reproduction, verification, and further research, the code for this paper is available at https://github.com/mangoleaves/OpenMeshCraft, and we provide a benchmark model list and a cookbook of our predicates in the supplementary material.

Datasets. We test our algorithm on the well-known Thingi10k dataset [Zhou and Jacobson 2016]. We filter out models with at least one intersection as our Thingi10k dataset and exclude the two most challenging models (ID.101633 and ID.996816) for the sake of a fair comparison (the statistics about the excluded models are individually reported in Table 1). The total count of models in the dataset is 4364. However, in the Thingi10k dataset, the number of intersecting triangle pairs in models typically ranges from 10^1 to 10^5 and mostly falls below 10^4 (as shown in the left part of Figure 12). This contrasts with the complexity of real industrial models, which often range from 10^4 to 10^7 (Figure 11), indicating a notable gap between the two. Therefore, we construct a more complex stress-testing dataset. We randomly select one model from the 4364 self-intersecting models, then make slight rotations (e.g., $[-3°, 3°]$) around three axes to obtain three rotated models, and finally, merge these four models into one self-intersecting model. We repeat the above operation to create 1000 models for our stress-testing dataset. The complexity of resulting models ranges from 10^4 to 10^7 and mostly falls below 10^6 (as shown in the right part of Figure 12). We validate our implementation by ensuring that our results consistently exhibit the same number of vertices and faces and Euler characteristics as those generated by competitors on the two datasets above.

Competitors. We compare our algorithm with the two most recent and publicly available algorithms [Cherchi et al. 2020; Zhou et al. 2016]. We obtain the latest implementation of [Zhou et al. 2016] from libigl [Jacobson et al. 2018] (version 2.5.0 in the GitHub repository), which has seen improved performance following an upgrade to CGAL's implementation of rational numbers [Hemmer et al. 2023]. The latest implementation of [Cherchi et al. 2020] (commit bf7eb71 in the GitHub repository), which integrates improvement from [Livesu et al. 2022] and [Cherchi et al. 2022], is now 3~8 times faster than the original implementation, as shown in [Cherchi et al. 2022]. The intersection resolution parts of both algorithms (`remesh_self_intersections` and `solveIntersections`, respectively) have the same problem settings as ours and are designed to resolve intersections exactly. In our tests, our implementation and Cherchi et al. [2020] succeed on all models while Zhou et al. [2016] fail on 40 models in the Thingi10k dataset and 46 models in the stress-testing dataset. These issues are primarily caused by runtime errors or exceeding the maximum memory limit (32 GB in our experiments), so we exclude these models when comparing with [Zhou et al. 2016].

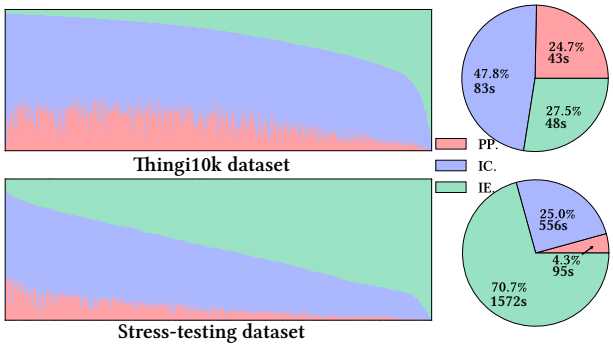


Fig. 13. The proportion of time for each part of our algorithm relative to the total time. The left figures display the time proportion across models, sorted by the intersection elimination time proportion for easier viewing. Meanwhile, the right pie charts show the time proportions of the whole dataset.

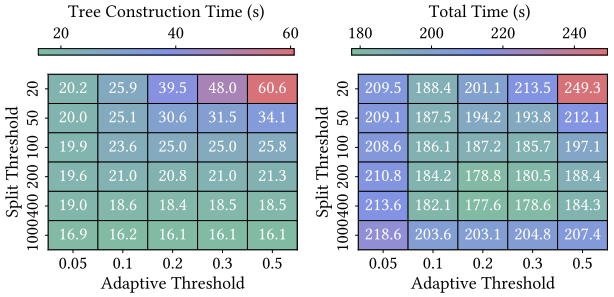


Fig. 14. Time of tree construction (left) and overall runtime of our algorithm (right) under various parameters of the adaptive tree. For comparison, with the adaptive threshold set to 1 (i.e., a simple OcTree) and the split threshold set to 1000, the tree construction time is 16.4 seconds, and the overall runtime is 293.9 seconds.

5.1 Evaluations

Timings and memory. We evaluate our algorithm on two datasets and analyze the runtime and peak memory usage through profiling. The distributions of these metrics are presented in Figure 12. Our algorithm exhibits nearly linear growth in runtime and memory usage as model complexity increases. On the Thingi10k dataset, each model's runtime is under 10 seconds, with only 18 models exceeding 1 second and an average runtime of 0.04 seconds across the dataset. For the stress-testing dataset, only 6 models require more than one minute, with an overall average runtime of 2.23 seconds. Regarding memory usage, peak usage reaches 3 GB and 21 GB on the Thingi10k and stress-testing datasets, with average usages of 53 MB and 960 MB, respectively. Our algorithm showcases the scalability of challenging models.

Performance breakdown. Our algorithm can be divided into three individual parts: (1) preprocessing routines and tree construction

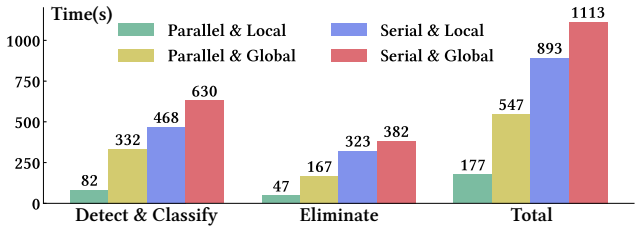


Fig. 15. Comparisons between global and local deduplication strategies on the Thingi10k dataset.

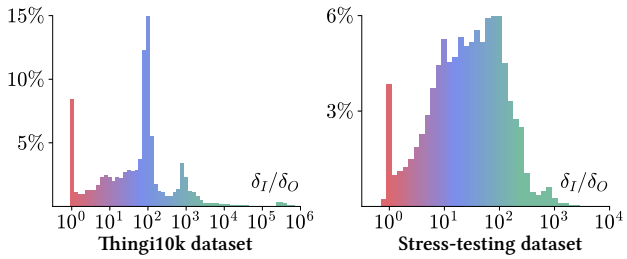


Fig. 16. The ratio of δ_I to δ_O in different datasets. We show the ratios via statistical histograms and assign gradient colors to facilitate distinction.

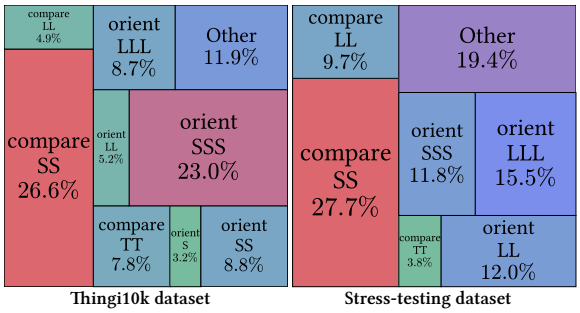
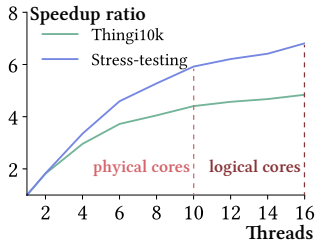


Fig. 17. The number of times that `pointCompare` and generalized `orient2d` are called with various combinations of implicit points. We abbreviate these two predicates as `compare` and `orient`. The combination of points is abbreviated by concatenating their respective abbreviations ("S" for LLI, "L" for LPI, and "T" for TPI). The explicit point is omitted in these abbreviations. For instance, when the input includes two LLI points (and despite how many explicit points), we label this combination as "SS". We only display combinations in which the proportion of calls exceeds 3% and group all other combinations under the "Other" category.

(PP.); (2) intersection detection and classification (IC.); (3) intersection elimination (IE.). In Figure 13, we show the time proportion for each part relative to the total time. It reveals that as the complexity of the model increases, the algorithm's bottleneck shifts from the intersection detection and classification step (shown in the upper) to the intersection elimination step (shown in the bottom).

Parallelism. We evaluate our algorithm with varying numbers of threads, from 1 to 16. For our serial execution (i.e., using only one thread), it takes about 15 minutes on the Thingi10k dataset and 262 minutes on the stress-testing dataset. In the inset, we present the speedup ratio of our algorithm with different numbers of threads compared to serial execution. The observed speedup is notable, especially for more challenging models, though it does not scale linearly due to memory bandwidth limitations, synchronization overhead, and parallelization costs.



Parameters of adaptive tree. In the adaptive tree, two parameters (the split and adaptive threshold) affect our algorithm's tree construction time and overall runtime. Our evaluations on the Thingi10k dataset, as shown in Figure 14, explore various combinations of these parameters. Setting the adaptive threshold to 1 makes the adaptive tree equivalent to a simple OcTree. For this, we only test with a split threshold of 1000, as smaller split thresholds prove too time-consuming due to too many redundant intersection tests. The analysis shows that although a smaller adaptive threshold and a larger split threshold effectively prevent over-partition and decrease the tree construction time, they increase the number of triangles in leaf nodes and, consequently, the time of intersection detection. Therefore, we choose the parameters with an adaptive threshold of 0.2 and a split threshold of 400.

Ablation studies. We first compare the local and global deduplication strategies on the Thingi10k dataset. The local strategy is detailed in Sections 4.1.2 and 4.2.2, and the global strategy sorts all points lexicographically within a global set, as used by [Cherchi et al. 2020]. We test our algorithm across four settings, comprising the combinations of serial or parallel execution with either local or global deduplication strategy (Figure 15). For serial execution, the local strategy proves faster than the global one due to its reduced sorting scale and complexity. For global strategy, the parallel execution shows less than a twofold speedup compared to its serial counterpart. In contrast, parallel execution using the local strategy demonstrates a notable speedup, as the local strategy effectively enhances parallelism by breaking down the sorting problem.

Then, we perform ablation on the simplified intersection location algorithm (Section 4.2.1). The ablation increases the time for the elimination step from 48 seconds to 128 seconds on the Thingi10k dataset, with the total time rising from 177 seconds to 258 seconds. For the stress-testing dataset, the elimination step's time increases from 1572 seconds to 5410 seconds, and the total time rises from 2230 seconds to 6086 seconds. As model complexity increases, the proportion of time spent on the elimination step also grows, demonstrating the increasing benefits brought by the algorithm.

Predicates. For semi-static filters of generated implicit points in two datasets, we compare the magnitude of inputs under indirect predicates and indirect offset predicates. We refer to the magnitude of inputs for all implicit points under the indirect predicates as δ_I and refer to their counterparts under the indirect offset predicates as δ_O. As shown in Figure 16, δ_I is generally several orders of magnitude higher than δ_O, leading to larger error bounds in semi-static filters under indirect predicates. We further count the number of times that `pointCompare` and generalized `orient2d` are called with various combinations of implicit points. We do not differentiate based on projection dimensions but focus on combinations of implicit points.

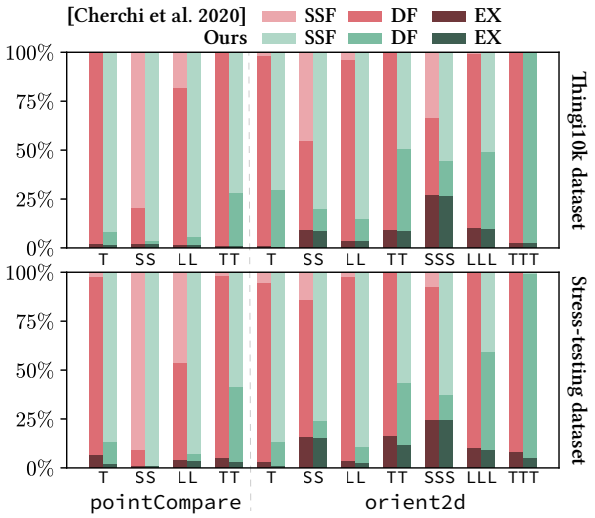


Fig. 18. The success rates of indirect predicates used in [Cherchi et al. 2020] (shaded in red) and our indirect offset predicates (shaded in green) at various filtering stages for different combinations of points. The abbreviations for the combinations of points follow the same convention described in Figure 17. The filtering stages contain semi-static filters (SSF), dynamic filters (DF), and floating-point expansions (EX), shaded in light, normal, and dark colors, respectively.

The results in Figure 17 demonstrate that LLI points participate extensively in predicate calls. Finally, we analyze the success rates of predicates at various filtering stages for different combinations of implicit points, showcasing the combinations with higher call frequencies in Figure 18. The results reveal that under the same combinations, indirect offset predicates tend to pass filter validation at earlier stages. Combinations including LLI points also pass the filter earlier than those containing LPI points. By introducing new implicit points and predicates, we achieve a total time reduction of 35% to 70% across two datasets.

5.2 Comparisons

Comparison of timings and memory. We conduct a comparison of runtime and memory usage between our algorithm and those proposed by [Zhou et al. 2016] and [Cherchi et al. 2020], with the results detailed in Figure 12. For the overall runtime, our algorithm outperforms [Cherchi et al. 2020] by approximately 15 times on the Thingi10k dataset and 30 times on the stress-testing dataset and outperforms [Zhou et al. 2016] by about 56 times and 75 times on these respective datasets. For runtime on each model, our algorithm is slower than [Cherchi et al. 2020] on only 45 models in the Thingi10k dataset. These models are extremely simple, with fewer than 100 intersecting triangle pairs, and their total runtime is merely 0.06 seconds. Our algorithm's memory usage is comparable to that of [Cherchi et al. 2020] and is much less than that of [Zhou et al. 2016]. In summary, our algorithm demonstrates superior runtime performance across both datasets, and its superiority is especially evident in more complex and challenging models. Moreover, our serial implementation is also 2.8 times faster than [Cherchi et al. 2020]'s parallel implementation on the Thingi10k dataset and 4 times faster on the stress-testing dataset. This highlights our contributions in both reducing algorithmic complexity and enhancing parallelism.

Table 1. Statistics of the ten most challenging models in the Thingi10k dataset. We rank models based on the scale of intersections and select the top 10 most challenging models. We then sequentially present the runtime and memory usage for all methods in the format "Ours/[Cherchi et al. 2020]/[Zhou et al. 2016]". Moreover, we show the proportion of the comparison methods' data compared to our method's data, indicated as "Ratio of [Cherchi et al. 2020]to ours/Ratio of [Zhou et al. 2016] to ours". "F" in the table indicates the failure of the algorithm.

Model ID	Runtime(s)	Runtime ratio	Memory (MB)	Memory ratio
252784	4.1/68.6/447.5	16.7/109.1	2395/1260/13753	0.5/5.7
101633	310.3/F/2721.0	F/8.8	1691/F/9170	F/5.4
55928	2.9/61.6/352.1	21.2/121.4	724/441/5581	0.6/7.7
1368052	4.3/164.7/452.8	38.3/105.3	3218/1968/15143	0.6/4.7
996816	253/2865.3/F	11.3/F	21872/15597/F	0.7/F
498461	1.0/15.6/101.2	15.6/101.18	538/270/2768	0.5/5.1
338910	0.6/5.5/67.9	9.2/113.25	535/246/2732	0.5/5.1
252785	0.8/16.3/67.7	20.4/84.6	475/271/2537	0.6/5.4
498460	0.5/10.7/70.1	21.4/140.2	488/264/2311	0.5/4.7
242236	0.6/15.7/168.8	26.2/281.3	540/600/2744	1.1/5.1

Performance across varying complexities. In Figure 20, we compare the performance of our algorithm with [Cherchi et al. 2020] across models of varying complexities. For simpler models, the computational hotspots are primarily in preprocessing and intersection detection, resulting in a relatively small speed advantage for our algorithm. As model complexity increases, the computational hotspots shift towards intersection classification and elimination, where our algorithm's superiority becomes increasingly apparent, often achieving speed-ups of an order of magnitude or more. In Figure 19, we categorize the models into five groups based on the runtime of our algorithm on them. From the simplest group to the most challenging group, the speed of our algorithm relative to [Cherchi et al. 2020] increases from 4 times to 30 times. We list the statistics for the ten most challenging models in the Thingi10k dataset, arranged according to models' complexities, in Table 1. The table reveals a conclusion consistent with the analysis above.

6 CONCLUSION AND DISCUSSION

We propose an approach to significantly enhance the efficiency of exact intersection resolution. The key contributions include a new concept of geometric predicates, a new type of implicit point, and localization and dimension reduction techniques for sorting, deduplicating, and locating the intersections. Rigorous testing against state-of-the-art algorithms demonstrates superior performance improvement, increasing speed by about 15~30 times. Statistical data analysis and detailed experiments validate the utility of our contributions, collectively making our algorithm a promising solution for intersection resolution in mesh arrangements.

Triangulation quality. We adopt the simple and efficient linearized earcut algorithm for triangulation. In contrast to our option, previous works [Attene 2014; Barki et al. 2015; Zhou et al. 2016] prefer constrained Delaunay triangulation (CDT) [Shewchuk and Brown 2015] to avoid poorly shaped triangles. A simple comparison between earcut triangulation and CDT is shown in Figure 21. Both triangulations have poorly shaped triangles. Therefore, improving

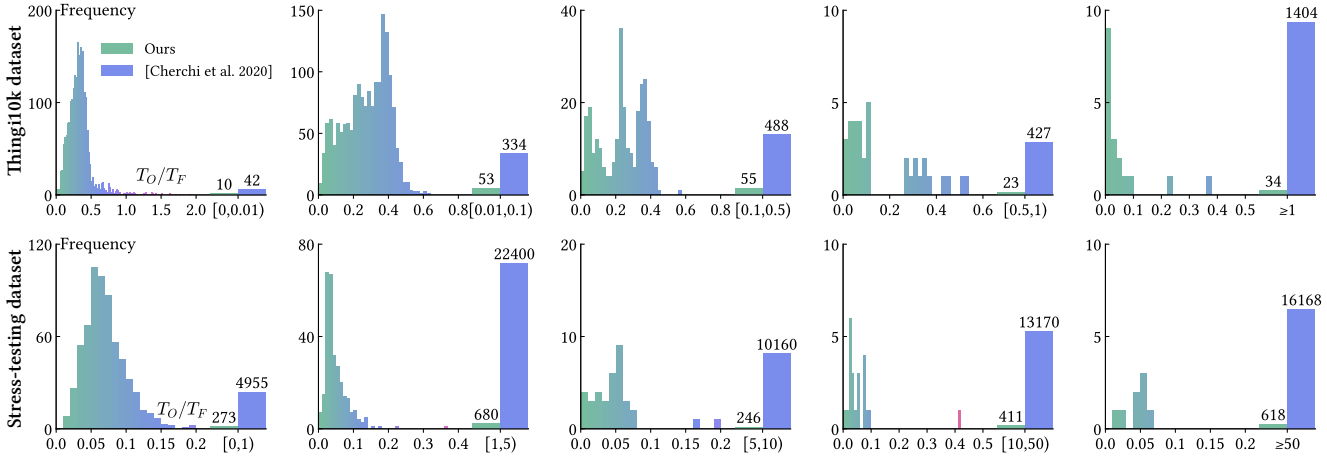


Fig. 19. Comparisons between our algorithm's runtime T_O and Cherchi et al. [2020]'s runtime T_F on the two datasets across varying complexities. The models are grouped into five categories based on our algorithm's runtime. For the Thingi10k dataset, the groups are $[0s, 0.01s)$, $[0.01s, 0.1s)$, $[0.1s, 0.5s)$, $[0.5s, 1s)$, $[1s, +\infty]$ from left to right in the upper row. For the stress-testing dataset, the groups are $[0s, 1s)$, $[1s, 5s)$, $[5s, 10s)$, $[10s, 50s)$, $[50s, +\infty]$ from left to right in the lower row. The horizontal axis represents the ratio of our algorithm's runtime to [Cherchi et al. 2020]'s runtime, i.e., T_O/T_F, with a lower value indicating that our algorithm is faster. The left plots display the frequency corresponding to the runtime ratio, while the right bars illustrate the total runtime of the two algorithms within each group. The numbers above the bars are the runtime in seconds.

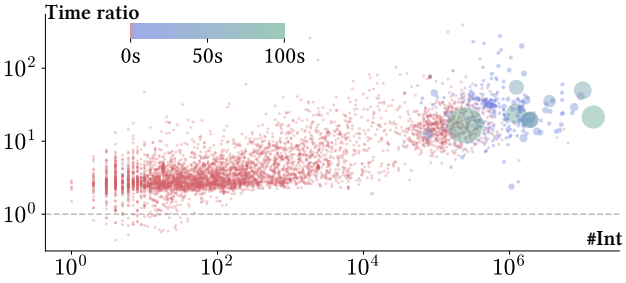


Fig. 20. Performance comparison of our algorithm with [Cherchi et al. 2020] across models of varying complexities. We present a scatter plot showing the ratio of the runtime of our algorithm to [Cherchi et al. 2020] vs. the number of intersecting triangle pairs. Each scatter point's radius is proportional to its corresponding runtime, with point colors mapped from red through blue to green based on the runtime.

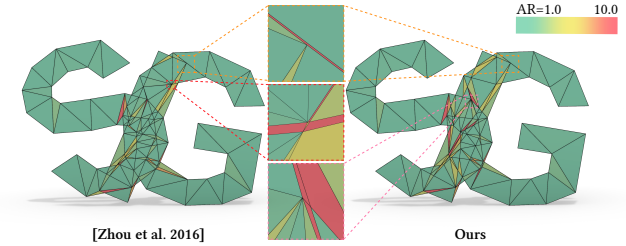


Fig. 21. Comparing the constrained Delaunay triangulation in [Zhou et al. 2016] with the earcut triangulation in our algorithm. Both triangulation results have 172 triangles. We calculate the aspect ratio (AR) for all triangles and map the aspect ratios to colors (the color bar is shown in the upper right corner). We zoom out the parts with poorly shaped triangles.

triangulation quality while ensuring efficiency is one of our future works.

Implicit point and predicates. The complexity of TPI points remains a challenge, with low success rates in filters containing TPI points (Figure 18) and computational bottlenecks due to floating-point expansion calculations on TPI points. Nehring-Wirxel et al. [2021] limit input precision and round floating-point numbers to a fraction of integers to maintain the precision of triplet plane intersection points' coordinates within a lower range, thus speeding up computations. Yet, to the best of our knowledge, no similar feature has been found for implicit points under IEEE-754 double-precision inputs. Simplifying implicit points and their associated computations remains a challenging and interesting problem for future research. Nevertheless, the proposed indirect offset predicates may enhance the efficiency of existing algorithms (e.g., conformal meshing [Diazzi and Attene 2021]), especially for those who do not rely on too complex implicit points, like constrained Delaunay triangulation [Diazzi et al. 2023].

Specific applications. Specific applications might impose further assumptions or constraints and develop tailored acceleration strategies based on their unique characteristics [Douze et al. 2017; Sheng et al. 2018; Trettner et al. 2022]. For example, a boolean algorithm might assume that triangles are derived from N solids or piecewise-constant integer generalized winding number (PWN) meshes and tailor pruning strategies to the specifics of boolean operations. These typically do not conflict with the basic problem we are addressing. Therefore, our algorithm, developed for the general purpose, is adaptable and may be improved to fit new problem settings.

Iterative operations. Iterative operations are required by some applications, meaning that the algorithm's output serves as part of the input for the next operations. Our current implementation does not adequately support this requirement, as it only accepts floating point numbers as input. This necessitates rounding the output of our algorithm before it can be reused as input, leading to numerical errors. Modifying our algorithm to accept implicit points as input

is possible. This would involve extending the `orient3d` predicate to support implicit points [Attene 2020] for detecting intersections and using explicit geometric primitives that construct input implicit points to construct new ones during classifying intersections. The only modification in the triangulation would be treating each triangle's input vertices as implicit points. Our algorithm can be extended to support iterative operations without introducing any new intersection point types, predicates, or techniques.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive suggestions and comments. This work is partially supported by the National Natural Science Foundation of China (62272429). Xiao-Ming Fu is a USTC Tang Scholar.

REFERENCES

Marco Attene. 2014. Direct repair of self-intersecting meshes. *Graphical Models* 76, 6 (2014), 658–668.
Marco Attene. 2020. Indirect Predicates for Geometric Constructions. *Comput. Aided Des.* 126 (2020), 102856.
Marco Attene, Marcel Campen, and Leif Kobbelt. 2013. Polygon Mesh Repairing: An Application Perspective. *ACM Comput. Surv.* 45, 2 (2013), 1–33.
Marco Attene, Marco Livesu, Sylvain Lefebvre, Thomas Funkhouser, Szymon Rusinkiewicz, Stefano Ellero, Jonàs Martínez, and Amit Haim Bermano. 2018. *Design, representations, and processing for additive manufacturing*. Vol. 10. Springer.
Hichem Barki, Gaël Guennebaud, and Sebti Foufou. 2015. Exact, robust, and efficient regularized Booleans on general 3D meshes. *Computers & Mathematics with Applications* 70, 6 (2015), 1235–1254.
Amit H Bermano, Thomas Funkhouser, and Szymon Rusinkiewicz. 2017. State of the art in methods and representations for fabrication-aware design. *Comput. Graph. Forum* 36, 2 (2017), 509–535.
Gilbert Bernstein and Don Fussell. 2009. Fast, Exact, Linear Booleans. *Comput. Graph. Forum* 28, 5 (2009), 1269–1278.
Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. 2010. *Polygon mesh processing*. CRC press.
Mario Botsch, Mark Pauly, Christian Rossl, Stephan Bischoff, and Leif Kobbelt. 2006. Geometric Modeling Based on Triangle Meshes. In *ACM SIGGRAPH 2006 Courses* (Boston, Massachusetts) *(SIGGRAPH '06)*. 1–es.
Mario Botsch and Olga Sorkine. 2007. On linear variational surface deformation methods. *IEEE. T. Vis. Comput. Gr.* 14, 1 (2007), 213–230.
Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. 1998. Interval Arithmetic Yields Efficient Dynamic Filters for Computational Geometry. In *Proc. Annu. Symp. Comput. Geom. (SCG '98)*. 165–174.
Hervé Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Stefan Schirra. 2024. 2D and 3D Linear Geometry Kernel. In *CGAL User and Reference Manual* (5.6.1 ed.). CGAL Editorial Board. https://doc.cgal.org/5.6.1/Manual/packages.html#PkgKernel23
Marcel Campen and Leif Kobbelt. 2010. Exact and Robust (Self-)Intersections for Polygonal Meshes. *Comput. Graph. Forum* 29, 2 (2010), 397–406.
Siu-Wing Cheng, Tamal Krishna Dey, Jonathan Shewchuk, and Sartaj Sahni. 2013. *Delaunay mesh generation*. CRC Press Boca Raton.
Gianmarco Cherchi, Marco Livesu, Riccardo Scateni, and Marco Attene. 2020. Fast and robust mesh arrangements using floating-point arithmetic. *ACM Trans. Graph.* 39, 6 (2020), 1–16.
Gianmarco Cherchi, Fabio Pellacini, Marco Attene, and Marco Livesu. 2022. Interactive and Robust Mesh Booleans. *ACM Trans. Graph.* 41, 6 (2022), 1–14.
Olivier Devillers, Sylvain Lazard, and William Lenhart. 2018. 3D Snap Rounding. In *Proc. Annu. Symp. Comput. Geom.* 1–14.
Olivier Devillers and Sylvain Pion. 2003. Efficient Exact Geometric Predicates for Delaunay Triangulations. *Proc. 5th Workshop Algorithm Eng. Exper.*, 37–44.
Lorenzo Diazzi and Marco Attene. 2021. Convex polyhedral meshing for robust solid modeling. *ACM Trans. Graph.* 40, 6 (2021), 1–16.
Lorenzo Diazzi, Daniele Panozzo, Amir Vaxman, and Marco Attene. 2023. Constrained Delaunay Tetrahedrization: A Robust and Practical Approach. *ACM Trans. Graph.* 42, 6 (2023), 1–15.
Matthijs Douze, Jean-Sébastien Franco, and Bruno Raffin. 2017. QuickCSG: Fast Arbitrary Boolean Combinations of N Solids. https://doi.org/10.48550/arXiv.1706.01558
S Fortune. 1999. Vertex-Rounding a Three-Dimensional Polyhedral Subdivision. *Discrete & Computational Geometry* 22, 4 (1999), 593–618.
Steven Fortune and Christopher J. Van Wyk. 1993. Efficient Exact Arithmetic for Computational Geometry. In *Proc. Annu. Symp. Comput. Geom. (SCG '93)*. 163–172.
Steven Fortune and Christopher J. Van Wyk. 1996. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.* 15, 3 (1996), 223–248.
Michael Hemmer, Susan Hert, Sylvain Pion, and Stefan Schirra. 2023. Number Types. In *CGAL User and Reference Manual* (5.6 ed.). CGAL Editorial Board. https://doc.cgal.org/5.6/Manual/packages.html#PkgNumberTypes
Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo. 2018. Tetrahedral meshing in the wild. *ACM Trans. Graph.* 37, 4 (2018), 1–14.
Alec Jacobson, Daniele Panozzo, et al. 2018. libigl: A simple C++ geometry processing library. https://libigl.github.io/
Mioara Joldes, Olivier Marty, Jean-Michel Muller, and Valentina Popescu. 2016. Arithmetic Algorithms for Extended Precision Using Floating-Point Expansions. *IEEE Trans. Comput.* 65, 4 (2016), 1197–1210.
Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. 2019. ABC: A Big CAD Model Dataset for Geometric Deep Learning. (2019), 9593–9603.
David H. Laidlaw, W. Benjamin Trumbore, and John F. Hughes. 1986. Constructive Solid Geometry for Polyhedral Objects. *SIGGRAPH Comput. Graph.* 20, 4 (1986), 161–170.
Bruno Lévy. 2016. Robustness and efficiency of geometric programs: The Predicate Construction Kit (PCK). *Comput. Aided Des.* 72 (2016), 3–12.
M. Livesu, G. Cherchi, R. Scateni, and M. Attene. 2022. Deterministic Linear Time Constrained Triangulation using Simplified Earcut. *IEEE. T. Vis. Comput. Gr.* 28, 12 (2022), 5172–5177.
Marco Livesu, Stefano Ellero, Jonàs Martínez, Sylvain Lefebvre, and Marco Attene. 2017. From 3D models to 3D prints: an overview of the processing pipeline. *Comput. Graph. Forum* 36, 2 (2017), 537–564.
Pion Meyer. 2008. FPG A code generator for fast and certified geometric predicates. *Real Numbers and Computers* (2008).
Julius Nehring-Wirxel, Philip Trettner, and Leif Kobbelt. 2021. Fast Exact Booleans for Iterated CSG using Octree-Embedded BSPs. *Comput. Aided Des.* 135 (2021), 103015.
Sang C Park. 2004. Triangular mesh intersection. *The Vis. Comput.* 20 (2004), 448–456.
Bin Sheng, Bowen Liu, Ping Li, Hongbo Fu, Lizhuang Ma, and Enhua Wu. 2018. Accelerated robust Boolean operations based on hybrid representations. *Comput. Aided Geom. Des.* 62 (2018), 133–153.
Johnathan Richard Shewchuk. 1996. *Robust Adaptive Floating-Point Geometric Predicates*. Association for Computing Machinery, 141–150.
Jonathan Richard Shewchuk. 1997. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete Comput. Geom.* 18, 3 (1997), 305–363.
Jonathan Richard Shewchuk and Brielin C. Brown. 2015. Fast segment insertion and incremental construction of constrained Delaunay triangulations. *Computational Geometry* 48, 8 (2015), 554–574.
Hang Si. 2015. TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator. *ACM Trans. Math. Software* 41, 2 (2015), 1–36.
Olga Sorkine. 2006. Differential representations for mesh processing. *Comput. Graph. Forum* 25, 4 (2006), 789–807.
Stratasys. 2024. GrabCAD. https://grabcad.com/library
Philip Trettner, Julius Nehring-Wirxel, and Leif Kobbelt. 2022. EMBER Exact Mesh Booleans via Efficient & Robust Local Arrangements. *ACM Trans. Graph.* 41, 4 (2022), 1–15.
Charlie C.L. Wang and Dinesh Manocha. 2013. Efficient Boundary Extraction of BSP Solids Based on Clipping Operations. *IEEE. T. Vis. Comput. Gr.* 19, 1 (2013), 16–29.
Weiming Wang, Tuanfeng Y. Wang, Zhouwang Yang, Ligang Liu, Xin Tong, Weihua Tong, Jiansong Deng, Falai Chen, and Xiuping Liu. 2013. Cost-effective Printing of 3D Objects with Skin-Frame Structures. *ACM Trans. Graph.* 32, 5 (2013), 1–10.
Z. Zheng, X. Gao, Z. Pan, W. Li, P. S. Wang, G. Wang, and K. Wu. 2024. Visual-Preserving Mesh Repair. *IEEE Trans Vis Comput Graph* PP (2024).
Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh arrangements for solid geometry. *ACM Trans. Graph.* 35, 4 (2016), 1–15.
Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016).
Afra Zomorodian and Herbert Edelsbrunner. 2000. Fast software for box intersections. In *Proc. Annu. Symp. Comput. Geom.* 129–138.