Robust and Efficient Local Repair for Intersecting Triangle Meshes

Taoran Liu* Hongfei Ye* Xiangqiao Meng† Zhiwei Liu* Jianjun Chen‡

Abstract

Triangle meshes often suffer from defects like intersecting triangles and low-quality elements. Existing intersection resolution methods either lack robustness due to floating-point inaccuracies or incur high computational costs by processing meshes globally. We propose a robust and efficient method for repairing meshes with intersecting triangles that combines localized processing with rational number computations. Our approach begins with a preprocessing step that refines the mesh and localizes intersection issues by separating intersecting and non-intersecting regions. For each intersecting region, we ensure the robustness of intersection calculations by using rational numbers. Subsequently, the intersection results are stably converted from rational to floating-point representation using a constrained boundary volumetric mesh optimization method. The repaired local meshes are then stitched back into the non-intersecting mesh, followed by a remeshing step to enhance overall mesh quality. Experimental results on complex models demonstrate that our method significantly reduces computational overhead while producing high-quality, non-intersecting meshes suitable for downstream applications.

1 Introduction

Triangle meshes are one of the fundamental geometric representations in computer graphics, widely utilized in areas such as 3D modeling, simulation, and animation. However, triangle meshes generated from various sources often contain defects, such as intersecting triangles and low-quality elements. These defects impede downstream applications, notably constrained Delaunay tetrahedrization [11], which is essential in computer graphics and engineering for tasks such as solving partial differential equations [24] and algorithms requiring explicit volume discretization [9, 20]. However, addressing mesh intersection issues poses significant challenges to algorithmic robustness due to the limitations of floating-point precision. Consequently, developing efficient and robust methods to resolve intersections and generate high-quality surface meshes represented with floating-point numbers has become a critical challenge.

Existing intersection resolution methods can be primarily categorized into two types. The first category performs intersection detection directly on the surface mesh [1, 10, 7, 14]. To mitigate the impact of floating-point precision errors, some researchers employ indirect predicates to represent intersection points. Although this approach can efficiently produce meshes that do not intersect when using indirect predicate representations, converting them to floating-point numbers for downstream applications often introduces new intersections due to floating-point inaccuracies.

The second strategy reformulates the triangle intersection computation as a boundary-conforming tetrahedral meshing problem [27, 18, 17]. Originally proposed by Xiao et al. [27], this approach aims to improve the efficiency of intersection calculations. However, due to inherent floating-point errors, the robustness of the algorithm cannot be fully guaranteed. To address this issue, Hu et al. [18] introduced rational numbers for conformal boundary recovery, mitigating the limitations of floating-point precision. Subsequently, to enable stable conversion of rational numbers to floating-point representations during volumetric mesh optimization, it is essential to ensure that no negative-volume tetrahedra are generated. However, this method requires processing all surface meshes uniformly, significantly increasing computational costs, as intersection operations are performed even in intersection-free regions.

To address this, we propose a method that combines local processing with rational number computation, aiming to balance intersection efficiency with the accuracy of floating-point results, thus achieving robust and efficient intersection results in floating-point representation. Specifically, we localize the intersection problem through a preprocessing step and extract local intersecting regions. For each local intersecting region, we introduce a constrained edge-based fast intersection method grounded in rational numbers, combined with volumetric mesh optimization to stably convert rational numbers to floating-point representations, thereby obtaining repaired local meshes. These repaired local meshes are then stitched with non-intersecting, correct

*Zhejiang University
†Hong Kong Polytechnic University
‡Corresponding author, chenjj@zju.edu.cn, Zhejiang University

meshes, followed by intersection-free remeshing to improve overall mesh quality. The final output is a high-quality, non-intersecting surface mesh suitable for downstream applications.

Our main contributions can be summarized as follows:

1. Proposed a Fast and Robust Intersection Repair Framework: We introduce a framework that takes low-quality, intersecting surface meshes as input and outputs high-quality, non-intersecting surface meshes. The core of the framework lies in the development of a robust and fast local intersection method based on rational numbers, which can effectively obtain intersection results represented in floating-point numbers.

2. Developed a Preprocessing Strategy to Localize Intersection Issues: By employing a connected component-based separation strategy, we efficiently segregate intersecting and non-intersecting meshes, reducing computational overhead and facilitating subsequent localized processing, thereby improving the overall efficiency of the algorithm.

2 Relate work

Resolving intersection issues in meshes is a fundamental challenge in computer graphics and geometric modeling. Over recent years, a variety of methods have been introduced to solve this problem. In this section, we provide an overview of the key approaches for addressing mesh intersections and remeshing.

Xiao et al. [27] proposed a novel algorithm that reformulates Boolean operations on triangulated solids as a boundary conforming tetrahedral meshing problem. However, the method encounters robustness issues due to floating-point errors. To address this, Hu et al.[18] introduced rational numbers to achieve precise intersection calculations. However, this approach suffers from low efficiency. To address these issues, Hu et al.[17] proposed an improved method known as Fast-TetWild. This method enhances computational speed by avoiding rational number operations and employing parallelization strategies. Nonetheless, Fast-TetWild still requires global boundary recovery and volumetric mesh optimization to improve mesh quality, resulting in considerable time overhead. Diazzi et al.[10] adopted a similar approach to that of Hu et al., but introduced indirect predicates[1] for boundary recovery in volumetric meshes. While this improved computational efficiency, it failed to robustly produce accurate results in floating-point representations. Skorkovská et al. [25] proposed a method that achieves high precision by carefully classifying all possible situations that could lead to imprecise floating-point calculations. Conor et al. [21] proposes a robust intersection algorithm for 2D planes using floating-point arithmetic. Portaneri et al.[23] introduced the Wrapping technique to resolve various intersection problems, but its capability to preserve geometric features is limited. Pion et al. [22] proposes a generic lazy evaluation scheme for exact geometric computations, which uses interval arithmetic to optimize the computation process and improve efficiency. The following methods introduced rely on this lazy evaluation. Cherchi et al.[6] proposed an innovative intersection method that does not explicitly compute intersection points but instead represents them using extended indirect predicates. While this approach yields valid intersection results under indirect predicates, it introduces new intersection issues when transitioning to floating-point numbers due to floating-point precision. Cherchi et al.[7] later improved this method to enhance computational efficiency, achieving real-time interactive Boolean operations. Similarly, Guo et al. [14] proposes a new geometric predicate, indirect offset predicate, and develops localization and dimension reduction techniques to boost efficiency and parallelism while maintaining accuracy. However, the problem of valid mesh representation in a floating-point representation remains unresolved. In summary, developing an efficient method to address model intersection problems in a floating-point representation remains a significant challenge.

Edge-based remeshing methods[15, 4] typically involve four local operations: edge splitting, edge collapsing, edge swapping, and vertex smoothing. Over the years, various adaptations of these methods have been developed to suit different remeshing applications by introducing constraints and optimizing strategies. Dunyach et al.[12] proposed a curvature-adaptive isotropic remeshing algorithm, which has demonstrated high efficiency in real-time applications. Dapogny et al.[8] introduced an adaptive remeshing algorithm for implicit geometries, which operates on constructed cubic Bézier surfaces and efficiently adapts to complex boundary changes. However, if the initial mesh quality is poor, issues such as folding and severe deformations may arise during the construction of these cubic Bézier surfaces. Hu et al.[16] proposed an error-bounded edge-based remeshing method that utilizes local operators to enhance mesh quality and reduce the number of elements while maintaining approximation error constraints. Cheng et al.[5] developed an improved error-bounded remeshing method that permits temporary violations of approximation error constraints during the remeshing process and ensures compliance by detecting and refining meshes that violate the constraints. Guo et

al.[13] applied edge-based remeshing techniques to discretize CAD models, while Wang et al.[26] introduced a method that eliminates large and small angles in surface meshes. Zhang et al.[28] presented a simple yet effective edge-based remeshing method that incorporates hard constraints for greater control over the remeshing process.

3 Method

We present a robust and efficient method for repairing meshes with intersecting triangles, consisting of five main steps as illustrated in Figure 1.

3.1 Preprocessing The input mesh M may contain numerous sliver or oversized triangles that self-intersect, thereby increasing the computational cost of intersection algorithms. To simplify these calculations, we first preprocess the input mesh, yielding the refined mesh M_p. This preprocessing not only reduces the computational cost of intersection but also improves the locality of intersecting triangles, facilitating local repairs. We adopt an edge-based remeshing algorithm during preprocessing, which adjusts the input mesh under the guidance of a constant sizing field and the constraints of a surface envelope. The constant sizing field ensures mesh uniformity, while the surface envelope maintains geometric fidelity during remeshing. Additionally, during the preprocessing stage, we will address issues such as hole and gap repairs to ensure the input mesh becomes watertight.

The constraint of the surface envelope refers to the requirement in the remeshing algorithm that the modified triangles must lie within the surface envelope. However, precise calculations are computationally expensive [2]. Therefore, an approximate method is used for the envelope test, where the position of the modified triangle is determined by checking if sampled points lie within the surface envelope. The specific sampling and judgment methods are based on the envelope testing strategy proposed by Hu et al [18].

A *constant sizing field* h specifies a uniform desired edge length for all meshes. The default value of h is set to 1% of the length of the diagonal of the model's axis-aligned bounding box.

To preserve the geometric fidelity during remeshing, we introduce a *surface envelope* constraint. The surface envelope $\mathcal{E}$ defines a tolerance region centered around the original surface with a thickness of δ, as follows:

$$\mathcal{E} = \{x \in \mathbb{R}^3 \mid \exists y \in S, \|x - y\| \leq \delta\} \tag{3.1}$$

where S is the original surface and δ is a predefined tolerance threshold.

The core idea of the edge-based remeshing algorithm is to iteratively adjust the mesh structure by performing a series of edge split, edge collapse, edge swap, and vertex smooth operations on the mesh edges, so that the mesh gradually meets the preset size and quality requirements [12].

These local operations are iteratively applied under the constraint of the surface envelope until a preset number of iterations is reached. Through this algorithm, we improve mesh quality while maintaining geometric accuracy, simplify intersection computations, and provide a solid foundation for subsequent local repair operations, as illustrated in Figure 1.

Additionally, to ensure the mesh is watertight, we employ the method proposed by Botsch et al. [3] to repair gaps and holes following mesh remeshing. Any intersections arising during this process are swiftly handled locally in subsequent steps.

3.2 Partition of Intersecting Triangles In this section, our objective is to identify all intersecting triangles within a given preprocessed mesh M_p and organize them into multiple connected components, while also categorizing the non-intersecting triangles into a separate set. To achieve this, we follow the procedure outlined below.

Let M_p denote the preprocessed triangular mesh, consisting of a set of triangles:

$$M_p = \{T_i \mid i = 1, 2, \ldots, N\}, \tag{3.2}$$

where N is the total number of triangles. Mesh M_p represents the preprocessed version of the original mesh, prepared for further analysis and intersection detection.

Based on the geometric information of mesh M_p, we construct an octree $\mathcal{O}$. Each leaf node ℓ of the octree $\mathcal{O}$ contains a subset of triangles from M_p, denoted as:

$$\mathcal{T}_\ell = \{T_i \in M_p \mid T_i \text{ is contained in leaf node } \ell\}. \tag{3.3}$$

For each leaf node ℓ, we perform pairwise intersection tests among all possible pairs of triangles (T_j, T_k) within the triangle set $\mathcal{T}_\ell$. These intersection tests are executed in parallel across different leaf nodes to enhance computational efficiency.

We define the set of all intersecting triangle pairs $\mathcal{I}$ as:

$$\mathcal{I} = \bigcup_\ell \{(T_j, T_k) \mid T_j, T_k \in \mathcal{T}_\ell,\ T_j \text{ intersects } T_k\}. \tag{3.4}$$

We then define the set of all triangles involved in intersections $\mathcal{G}$ as:

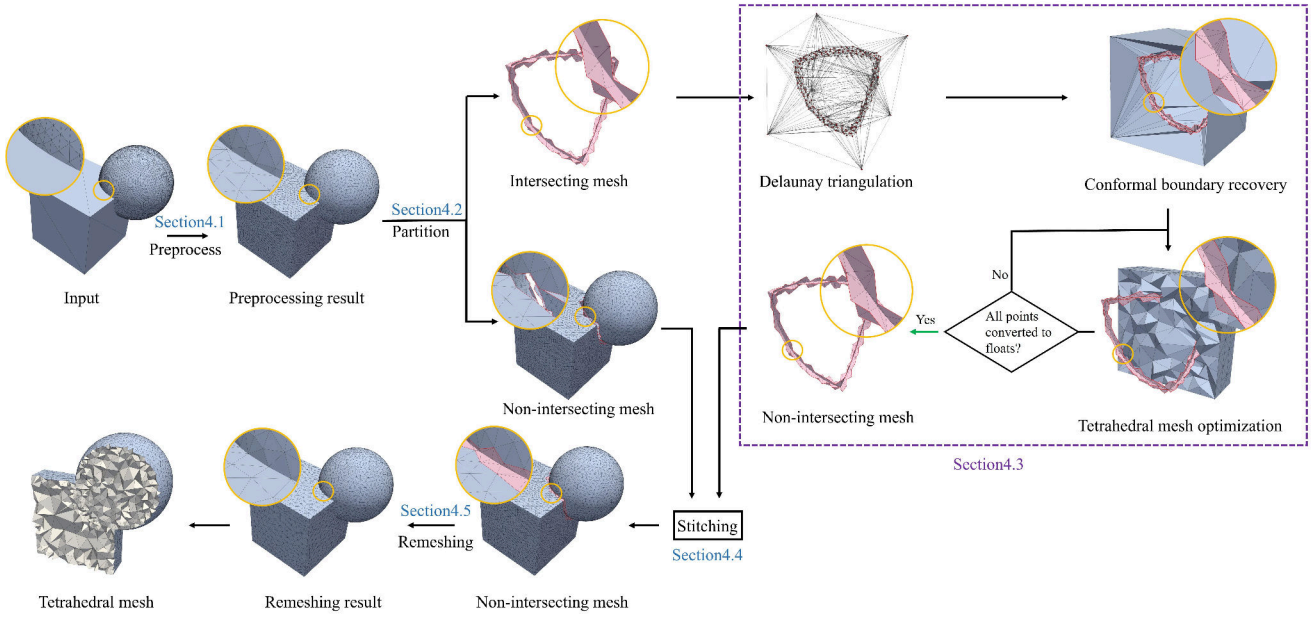


Figure 1: Pipeline of our method.

$$\mathcal{G} = \{T_i \in M_p \mid \exists\, T_j \in M_p,\ (T_i, T_j) \in \mathcal{I}\}. \tag{3.5}$$

In other words, $\mathcal{G}$ comprises all triangles in the preprocessed mesh M_p that intersect with at least one other triangle.

We construct an undirected graph $G = (V, E)$, where, $V = \{v_i \mid T_i \in \mathcal{G}\}$, with each vertex v_i corresponding to an intersecting triangle T_i. $E = \{(v_i, v_j) \mid (T_i, T_j) \in \mathcal{I}\}$, where an edge (v_i, v_j) indicates that triangles T_i and T_j intersect.

By performing connected component analysis on graph G, we obtain K connected components (i.e., subgraphs):

$$G = \bigcup_{k=1}^{K} G_k, \tag{3.6}$$

where each connected component $G_k = (V_k, E_k)$ satisfies:

- $V_k \subset V$ and $E_k \subset E$,
- Any two vertices within V_k are connected by a path in G_k.

For each connected component G_k, we define the corresponding set of triangles S_{int}^k as:

$$S_{\text{int}}^k = \{T_i \in \mathcal{G} \mid v_i \in V_k\}, \quad k = 1, 2, \ldots, K. \tag{3.7}$$

Each set S_{int}^k contains all intersecting triangles within the connected component G_k.

We define the set of non-intersecting triangles S_{non} as:

$$S_{\text{non}} = M_p \setminus \mathcal{G} = \{T_i \in M_p \mid T_i \notin \mathcal{G}\}. \tag{3.8}$$

Thus, S_{non} comprises all triangles in the preprocessed mesh M_p that do not intersect with any other triangle.

Ultimately, we obtain a collection of sets:

$$\{S_{\text{non}}, S_{\text{int}}^1, S_{\text{int}}^2, \ldots, S_{\text{int}}^K\}, \tag{3.9}$$

where:

- S_{non} is the set of all non-intersecting triangles.
- Each S_{int}^k for $k \in K$ is a connected component consisting of intersecting triangles.

For both the non-intersecting set S_{non} and each intersecting set S_{int}^k, we can define the corresponding vertex sets and edge sets as follows:

Vertex Sets: The vertex set V_{non} of S_{non} and V_{int}^k of S_{int}^k represent all the distinct vertices of the triangles in each set:

$$V_{\text{non}} = \bigcup_{T_i \in S_{\text{non}}} \{v_{i_1}, v_{i_2}, v_{i_3}\}, \tag{3.10}$$

$$V_{\text{int}}^k = \bigcup_{T_i \in S_{\text{int}}^k} \{v_{i_1}, v_{i_2}, v_{i_3}\}, \tag{3.11}$$

where $v_{i_1}, v_{i_2}, v_{i_3}$ are the vertices of triangle T_i.

Edge Sets: The edge set E_{non} of S_{non} and E_{int}^k of S_{int}^k represent all the edges of the triangles in each set:

$$E_{\text{non}} = \bigcup_{T_i \in S_{\text{non}}} \{(v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), (v_{i_3}, v_{i_1})\}, \tag{3.12}$$

$$E_{\text{int}}^k = \bigcup_{T_i \in S_{\text{int}}^k} \{(v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), (v_{i_3}, v_{i_1})\}. \tag{3.13}$$

This framework effectively organizes the triangles in the preprocessed mesh M_p into sets of intersecting and non-intersecting triangles. By constructing an octree and employing graph-based connected component analysis, our method efficiently identifies and categorizes intersecting triangles into multiple connected domains. This partition establishes a foundation for subsequent geometric processing and mesh repair operations.

3.3 Intersection Algorithm To ensure the robustness of intersection operations, inspired by the TetWild method proposed by Hu et al.[18], we present a fast and robust intersection algorithm based on rational numbers that supports constrained edges. The pipeline of our algorithm is depicted in Figure 2, where we use a 2D example for clear illustration. In this visualization, the segments represent mesh facets, and their endpoints correspond to mesh edges. The algorithm begins by classifying input segments into non-intersecting (green) and intersecting (yellow) parts. This algorithm aims to quickly resolve intersection issues without considering mesh quality initially, and its results are represented using floating-point numbers. The inclusion of constrained edges ensures that the constrained boundaries of the repaired mesh are preserved, facilitating subsequent stitching with the non-intersecting set S_{non}.

Our method first computes the bounding box of the intersecting part S_{int}^i, and generates eight virtual vertices by slightly expanding the minimum and maximum points of the bounding box. These vertices are then added to the point set V_{int}^i. Based on V_{int}^i, we construct the initial tetrahedral mesh M_{Tet} using the Bowyer-Watson algorithm. This is demonstrated in the Delaunay triangulation step of Figure 2.

We sequentially insert the triangles in S_{int}^i into M_{Tet}^i to perform conforming boundary recovery, obtaining a conforming tetrahedral mesh $M_{\text{Tet}}^{i'}$. The insertion process utilizes a Binary Space Partitioning (BSP) tree, with all operations performed using rational arithmetic. During this stage, as illustrated in Figure 2, intersection points (red) are represented using rational numbers, while constrained points (blue) and other vertices (black) are represented in floating-point format. Since the intersection point is calculated using rational numbers, robustness is ensured.

Next, we attempt to convert the rational coordinates of the tetrahedral mesh $M_{\text{Tet}}^{i'}$ into floating-point numbers, ensuring that the volume of the tetrahedra remains positive throughout. As shown in Figure 2, this involves decomposing the convex polygon by connecting its centroid to all vertices, followed by local mesh optimization. If all conversions are successful, the intersection algorithm terminates, and we extract the surface mesh from the conforming volumetric mesh, as indicated by the green arrow in Figure 1.

If the conversion is not entirely successful, we proceed to perform constrained volumetric mesh optimization. We define the set of constrained edges E_{con}^i as the intersection of E_{int}^i and E_{non}, as shown by the red edges in Figure 1. The volumetric mesh $M_{\text{Tet}}^{i'}$ is optimized under the constraints of E_{con}^i and the minimal surface envelope to improve mesh quality. This optimization ensures a stable transition from rational numbers to floating-point representations while maintaining consistently positive mesh element volumes. The E_{con}^i constraint facilitates subsequent stitching with the non-intersecting mesh S_{non}, whereas the minimal surface envelope constraint prevents intersections between the repaired S_{int}^i and S_{non}. By default, the tolerance δ for the minimal surface envelope is set to 1×10^{-6} times the diagonal length of the axis-aligned bounding box of S_{int}^i. Once all rational numbers are successfully converted, the intersection process concludes.

During the optimization process, we track and mark the triangles that belong to the input surface, similar to the yellow segments shown in the final step of Figure 2. Once all points are successfully converted to floating-point numbers, we extract the marked surface as the final result of the mesh intersection.

Volume mesh optimization is performed by iteratively applying four local operations to improve mesh quality, as illustrated in Figure 3. During volumetric mesh optimization, only three types of operations are allowed on the constrained edges E_{con}^i: inserting a point on a constrained edge, merging two adjacent vertices on a constrained edge, and merging a point from a non-constrained edge onto a constrained edge, as illustrated in Figure 4. The overall process of our intersection method is outlined in Algorithm 3.1.

ALGORITHM 3.1. (ROBUST FAST INTERSECTION) Based on Rational Numbers

Require: Intersecting set S_{int}^i

Ensure: Non-intersecting Surface mesh S_{fix}^i

1: **function** ROBUST-FAST-INTERSECTION(S_{int}^i)

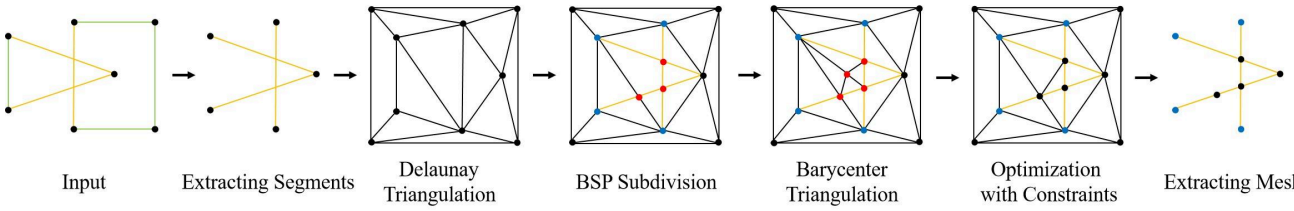


Figure 2: Pipeline of our intersection handling algorithm in 2D. The process includes: input segments classification (green: non-intersecting, yellow: intersecting), Delaunay triangulation of intersecting segments, BSP tree construction for intersection computation, polygon decomposition, mesh quality optimization, and final segment extraction.

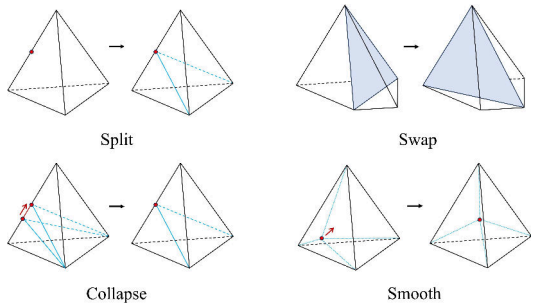


Figure 3: Four local operations.

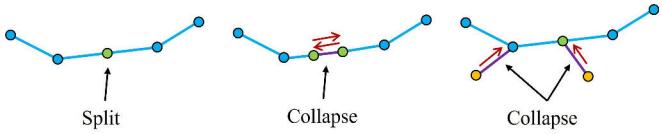


Figure 4: Diagram of optimization operations on constrained edges during volume mesh optimization. Blue points and edges represent segments of constrained edges. Green points indicate points inserted along the constrained edges. Purple edges are those being prepared for collapsing, while yellow points lie outside the constrained edges.

2: Compute bounding box of S^k_{int}, expand the minimum and maximum points slightly, and add the 8 virtual corner points to V^i_{int}
3: Construct tetrahedral mesh M_{Tet} using Bowyer-Watson on V^i_{int}
4: **for** each triangle $t \in S^i_{\text{int}}$ **do**
5: Insert t into M_{Tet} for boundary recovery
6: **end for**
7: Obtain conforming tetrahedral mesh M'_{Tet}
8: **if** Conversion of M'_{Tet} to floating-point is successful **then**
9: Extract surface mesh from M'_{Tet}
10: **return** Surface mesh S^i_{fix}
11: **else**
12: Define constrained edges $E^i_{\text{con}} = E^i_{\text{int}} \cap E_{\text{non}}$
13: Define the surface envelope $\mathcal{E}$ of S^i_{int}
14: **repeat**
15: Optimize M'_{Tet} subject to E^i_{con} and $\mathcal{E}$
16: Convert rational coordinates to floating-point
17: **until** Conversion complete
18: Extract surface mesh from optimized mesh
19: **return** Surface mesh S^i_{fix}
20: **end if**
21: **end function**

3.4 Stitching Since vertices were inserted into the constraint edge E^i_{con} during the volumetric mesh optimization, we need to insert corresponding points on the edges E^i_{non} in S_{non} during the stitching process to ensure successful merging. As shown in Figure 5, the red edges and blue points constitute E^i_{non}. The green points are the vertices inserted onto the constraint edges during the volumetric mesh optimization, and the red points are the vertices inserted on S_{non} for the purpose of stitching.

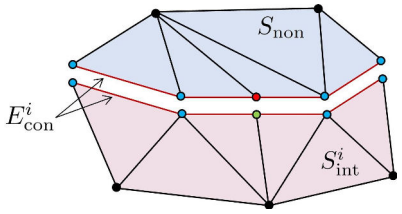


Figure 5: Alignment of inserted points prior to stitching.

Each S^i_{int} is stitched with S_{non}, forming the repaired surface mesh M_{fix}.

3.5 Remeshing While the repair process corrected geometric errors, it did not effectively improve the mesh quality. To improve the mesh quality, we perform an edge-based mesh remeshing that consists of two main steps: initializing the size field and remeshing the mesh

guided by this size field.

The size field $L(x)$ determines the desired edge lengths across the mesh, controlling mesh density. We initialize $L(x)$ based on the discrete curvature at each vertex. Specifically, for a vertex v_i, we compute the maximum absolute curvature κ_i using the principal curvatures $\kappa_{\max,i}$ and $\kappa_{\min,i}$:

$$\kappa_i = \max\left\{\left|\kappa_{\max,i}\right|, \left|\kappa_{\min,i}\right|\right\}. \tag{3.14}$$

Given an approximation tolerance ϵ, the size at vertex v_i is calculated as [12]:

$$L(v_i) = \sqrt{\frac{6\epsilon}{\kappa_i} - 3\epsilon^2}, \tag{3.15}$$

To reduce the impact of noise and ensure smooth transitions in the size field, we apply Laplacian smoothing to the size values at non-feature vertices. This process helps eliminate local irregularities that could lead to poor mesh quality.

After the construction of the size field, we iteratively improve the mesh quality using the four local operations described in Section 3.1. It is important to perform intersection checks to prevent the creation of new intersections during the remeshing process.

4 Experiments

Our approach was developed using standard C++ in Visual Studio 2022 on the Windows platform. The experiments were conducted on a desktop computer equipped with an AMD Ryzen 7 7840H processor and 32 GB of memory.

We selected the Ford Engine Block (#114029) model from the Thingi10K dataset [29] (as shown in Figure 6) and the Oil Pump Jack[1] model from the Meshing Contest (as shown in Figure 12). These models contain a large number of geometric errors and were used to verify the effectiveness of our method (see Section 4.1). We compared our method with the state-of-the-art Fast-TetWild [17] to validate its superiority (see Section 4.2). To ensure geometric fidelity, we set both the surface envelope tolerance δ during the mesh preprocessing phase and the approximation tolerance ϵ in the remeshing phase to 10^{-4} times the length of the diagonal of the model's axis-aligned bounding box.

4.1 Evaluation In this section, we use the Ford Engine Block model to verify the effectiveness of each algorithm step, demonstrating that each step plays a significant role in the overall processing. The Ford Engine Block model, as shown in Figure 6, contains numerous intersections, gaps and sliver triangles.

[1] https://grabcad.com/library/oil-pumpjack-1

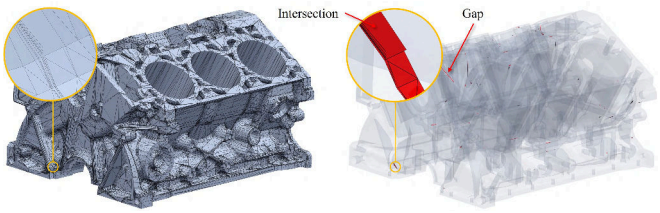


Figure 6: The Ford Engine Block model. This model contains 490 intersecting triangles and several gaps. In the right image, we set the model's transparency to 0.1 to highlight the gaps and intersecting triangles.

First, we present the results of the model preprocessing, as shown in Figure 7. After preprocessing, the model is watertight and contains only 61 intersecting triangles. The number of mesh intersections is significantly reduced. Furthermore, in regions where the mesh topology was correct but the geometry was erroneous, intersection issues were resolved after remeshing. This improvement is attributed to local operations such as smoothing during preprocessing, which performed untangling. Consequently, meshes that were originally intersecting became non-intersecting after remeshing, as indicated by the yellow box in the left image of Figure 7. These results verify the effectiveness of our preprocessing method.

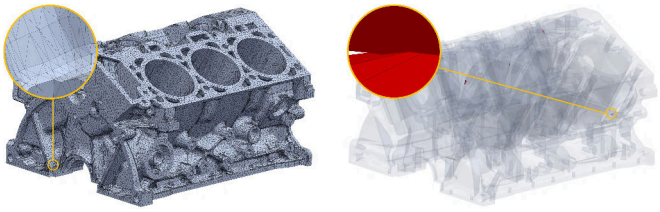


Figure 7: The model after preprocessing. In the right image, we set the model's transparency to 0.1 to highlight the intersecting triangles.

After preprocessing, we divide the model into different sets based on the partition algorithm introduced in Section 3.2. For each intersecting set, we introduced robust intersection operations based on rational numbers and utilized volumetric mesh optimization to gradually convert rational numbers to floating-point numbers. This process is valid as long as the tetrahedral mesh has positive volume. As shown in Figure 8, we demonstrate the partition results and the bounding box elements constructed for each intersecting set. A total of 11 intersecting sets were identified.

We display the non-intersecting surface mesh after repairing the intersecting sets and stitching them with the non-intersecting sets in Figure 9. We successfully

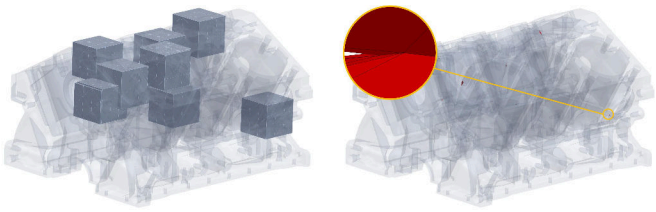


Figure 8: The local repair process. The left image shows multiple bounding box volumetric meshes constructed during local repair. In the right image, we set the model's transparency to 0.1 to highlight a repaired intersecting set.

generated the volumetric mesh using TetGen [24]. The three processes of partition, repair, and stitching were executed serially, taking only 4.04 seconds to complete the repair of the entire model. This demonstrates the advantage of our proposed local method, which significantly reduces the problem size.

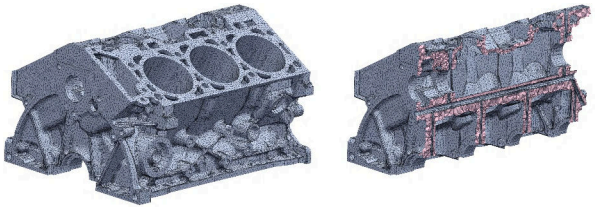


Figure 9: The repaired mesh. The left image shows the repaired surface mesh by our approach, and the right image shows a cross-section of the volumetric mesh generated using TetGen.

Finally, we employed remeshing to improve the mesh quality to meet the requirements of simulation analysis. The result of the remeshing is shown in Figure 10.

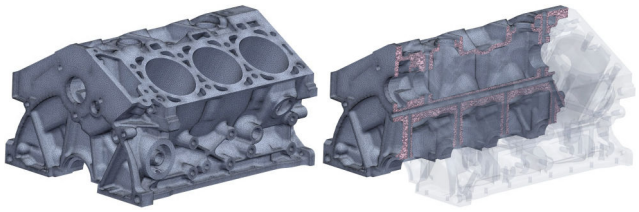


Figure 10: The surface mesh of the Ford Engine Block model generated by our method, and the volumetric mesh generated using TetGen.

To verify the robustness of our method, we further tested it using a more complex model, the Oil Pump Jack, as shown in Figure 12. This model consists of

Figure 11: The surface mesh of the Oil Pump Jack model generated by our method, and the volumetric mesh generated using TetGen.

114,524 triangles, with 48,528 triangles having intersection errors making it highly challenging.

Figure 12: The Oil Pump Jack model. This model contains 48,528 intersecting triangles. In the right image, we set the model's transparency to 0.1; red triangles represent triangles with intersection errors.

After preprocessing and partition, 25 intersecting sets were generated, as shown in Figure 13. The number of intersecting triangles was reduced to 38,111, approximately 10,000 fewer than the initial mesh, further verifying the effectiveness of our preprocessing step.

The final surface mesh generated by our method is shown in Figure 11, along with the volumetric mesh generated using TetGen. This verifies that the surface mesh produced by our method can successfully generate volumetric meshes.

Figure 14 presents a histogram illustrating the mesh quality in terms of skewness [19] for both the original model and the repaired model. The vertical axis represents the proportion, providing a quantitative insight into the distribution. Compared to the original mesh, the repaired mesh demonstrates superior quality, characterized by a higher prevalence of triangles approaching the ideal equilateral shape.

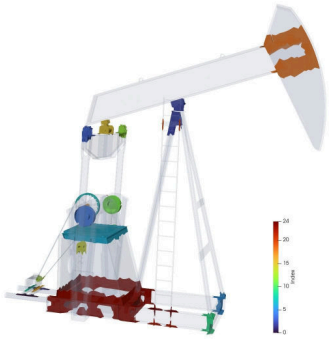
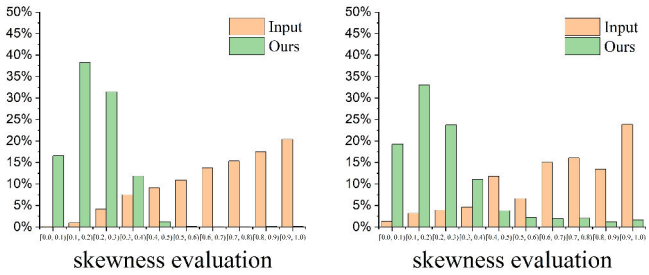


Figure 13: Schematic of intersecting sets after partition of the Oil Pump Jack model.



(a) Ford Engine Block (b) Oil Pump Jack

Figure 14: The comparison of mesh quality before and after remeshing for the Ford Engine Block and Oil Pump Jack model. Green: the result of original surface; Yellow: the repaired result by our method. The X-axis shows skewness values from 0.0 to 1.0 in intervals of 0.1, while the Y-axis shows the percentage of triangles in each interval.

4.2 Comparison We compared our method with Fast-TetWild [17], an improved version of TetWild [18] proposed by Hu et al. We chose Fast-TetWild as our comparison baseline because it represents the state-of-the-art method that achieves a balance between computational efficiency and result validity in floating-point representation, while other existing methods either struggle with floating-point accuracy or computational efficiency. We set the tolerance to 1×10^{-4} times the length of the model's bounding box diagonal, consistent with our method. Since Fast-TetWild with the default number of iterations results in excessively long computation times, causing it to remain in the optimization loop, we set its maximum number of iterations to 30. We present the results of Fast-TetWild processing the Ford Engine Block and Oil Pump Jack models in Figure 15.

We conducted comparisons focusing on local details, as illustrated in Figures 16 and 17. The Fast-TetWild

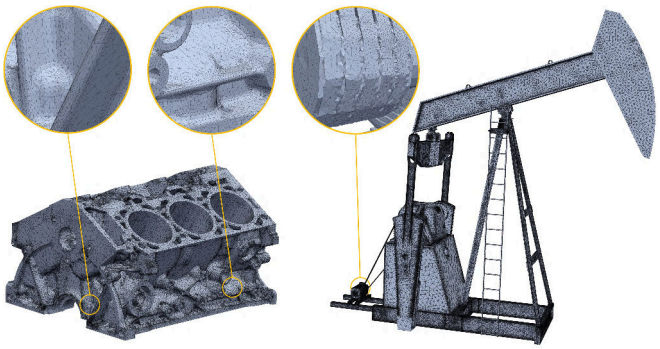
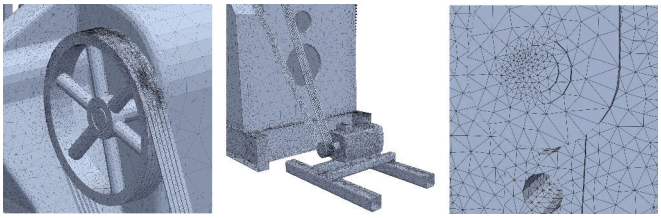
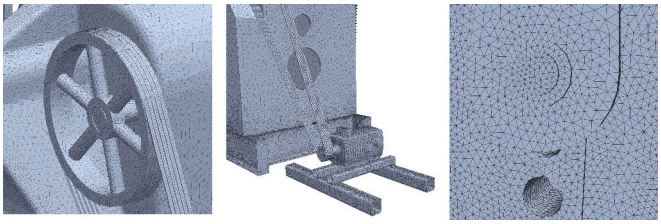


Figure 15: Results generated using the Fast-TetWild method.

exhibits local over-refinement, resulting from continuous refinement to improve quality. Our method only perform volumetric mesh optimization on locations with intersection, converting rational numbers to floating-point representations, and then use remeshing to improve mesh quality. This process can identify and preserve feature edges. In contrast, Fast-TetWild converts the entire model into a volumetric mesh for optimization, spending a significant amount of time on boundary recovery even in regions without intersection. Therefore, our method balances efficiency, robustness, mesh quality, and geometric fidelity.



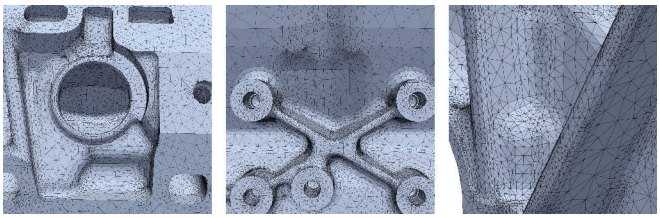
(a) Fast-TetWild



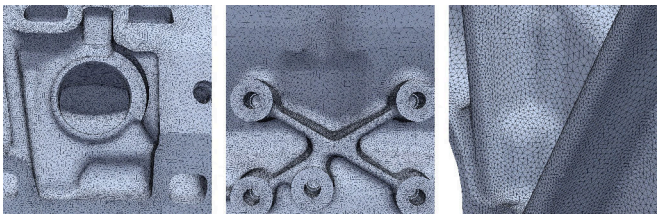
(b) Our method

Figure 16: Comparison of results on the Oil Pump Jack model between our method and Fast-TetWild.

We compared the running times of our method and Fast-TetWild in Table 1. Our method, executed serially, is still an order of magnitude faster than Fast-TetWild, demonstrating its high efficiency. It can be seen that the preprocessing time of our method on the Oil Pump



(a) Fast-TetWild



(b) Our method

Figure 17: Comparison of results on the Ford Engine Block model between our method and Fast-TetWild.

Table 1: Performance comparison between our method and Fast-TetWild

Model	Method	Mesh Stats.		Timing (s)			
		#Vert.	#Faces	Prep.	Inter.	Remesh	Total
Engine	Ours	916,773	1,834,123	51.76	4.04	107.52	**163.32**
	Fast-TetWild	387,006	774,452	–	–	–	9,858.09
Oil	Ours	517,517	1,073,913	17.74	537.14	128.46	**683.34**
	Fast-TetWild	497,239	1,038,747	–	–	–	7,114.36

Jack model is much shorter than on the Ford Engine Block model, because most of the mesh in the Oil Pump Jack model is located on planes, resulting in higher preprocessing efficiency.

Additionally, for the Ford Engine Block model, due to its fewer intersection errors and high locality (as shown in Figure 6), the intersection stage is very efficient, taking only 4.04 seconds. However, when processing the Oil Pump Jack model, the intersection algorithm required a significant amount of time due to the large number of intersections and their poor spatial locality. Despite this, it remained an order of magnitude faster than Fast-TetWild. We employed the method for fast and robust mesh arrangements using floating-point arithmetic, as proposed by Cherchi et al. [6], to handle intersections. Although the intersection is efficient, the result for the Oil Pump Jack model under floating-point representation still contains many intersecting triangles. Therefore, we believe that this intersection time is worthwhile to improve algorithm robustness.

4.3 Additional Validation on Diverse Geometries While the above analysis demonstrates the effectiveness of our method on complex CAD models, we further validate our approach on a broader range of geometries from the Thingi10K dataset [29]. In addition to the previously analyzed Engine, we randomly selected six more models with self-intersection defects to provide a comprehensive evaluation. For fair comparison, we maintained consistent parameter configurations as in the previous experiments.

Detailed visual results and comparisons can be found in Figure 19, which presents the processing results on these representative models: Gear (#34783), Yoda (#37861), Dragon (#39507), Turbine (#43149), Lamp (#61258), and Machine (#117682). Our method successfully handles these complex geometries while preserving geometric features. To verify the validity of our results, we use the processed surface meshes as boundary constraints to generate tetrahedral meshes, with cross-sections shown in pink to reveal the internal tetrahedral structure.

The quantitative evaluation further supports our previous findings. The angle distribution analysis (Figure 18) demonstrates that our method achieves superior mesh quality. The computational efficiency comparison (Table 2) consistently shows that our method significantly outperforms Fast-TetWild in processing speed, reinforcing our earlier observations.

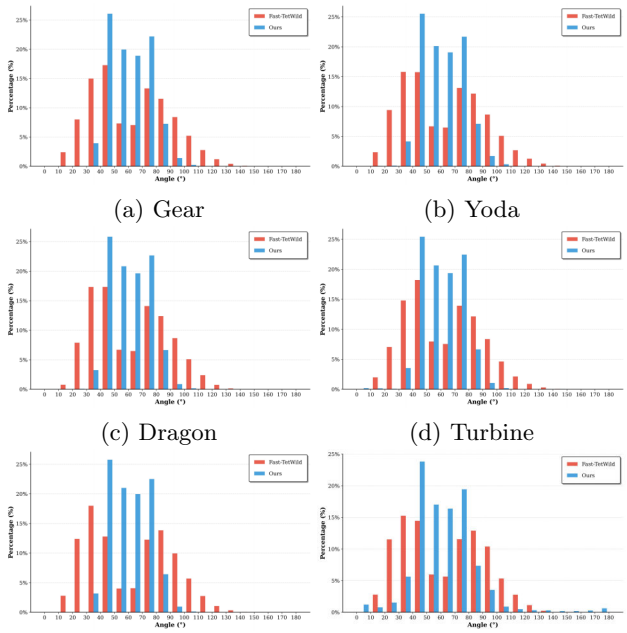
The consistent performance across both CAD models and diverse examples from the Thingi10K dataset demonstrates the robustness and generality of our method in handling various types of self-intersection defects while maintaining high efficiency and mesh quality.

Table 2: Performance comparison between our method and Fast-TetWild

Model	Method	Mesh Statistics		Timing (s)			
		#Vert.	#Faces	Prep.	Inter.	Remesh	Total
Gear	Ours	165,602	331,204	8.30	0.54	13.31	**22.15**
	Fast-TetWild	74,234	148,468	-	-	-	2532.50
Yoda	Ours	176,857	354,588	16.22	2.25	41.94	**60.41**
	Fast-TetWild	56,155	113,397	-	-	-	1070.93
Dragon	Ours	138,405	277,352	104.29	4.67	129.99	**238.95**
	Fast-TetWild	112,208	224,804	-	-	-	2075.04
Turbine	Ours	169,059	338,150	6.55	1.95	18.77	**27.27**
	Fast-TetWild	94,801	189,682	-	-	-	2923.17
Lamp	Ours	382,857	767,330	24.14	4.10	60.59	**88.83**
	Fast-TetWild	147,248	296,112	-	-	-	6838.84
Machine	Ours	309,837	694,389	13.22	217.02	280.68	**510.92**
	Fast-TetWild	76,437	156,621	-	-	-	3920.32

5 Conclusion

In this paper, we present a robust and efficient method for repairing triangle meshes with intersecting triangles



(a) Gear (b) Yoda

(c) Dragon (d) Turbine

(e) Lamp (f) Machine

Figure 18: Dihedral angle distributions of triangular meshes for different models generated by Ours and Fast-TetWild.

by combining localized processing with rational number computations. The process begins with a preprocessing step to improve the input mesh quality and enhance the locality of intersection calculations. Additionally, holes and gaps are repaired to ensure watertightness. We partition the mesh into intersecting and non-intersecting regions using octree-based spatial subdivision and connected component analysis. For intersecting regions, intersection calculations are made robust through the use of rational numbers. These results are then stably converted to floating-point representation via a constrained edge volumetric mesh optimization method. Finally, the repaired local meshes are seamlessly stitched into the non-intersecting mesh, followed by the application of an edge-based remeshing technique, guided by a curvature-adaptive sizing field, to improve overall mesh quality while avoiding intersections.

Experimental results on complex models such as the Ford Engine Block and Oil Pump Jack demonstrate that our method significantly reduces computational overhead while producing high-quality, intersection-free meshes suitable for downstream applications like finite element analysis. The localized nature of our method allows for significant time savings by avoiding unnecessary global processing, and the use of rational arithmetic ensures robustness against floating-point precision errors.

While our method demonstrates significant advantages in handling local intersections, it has certain limitations. In scenarios where dense intersections occur globally throughout the model, lacking locality characteristics, our method would need to repair the entire model and perform volumetric optimization, which could impact computational efficiency. Although such globally dense intersection scenarios are rare in practical applications. Additionally, our method relies on parameter settings, including mesh size for initial remeshing and surface envelope tolerance for tetrahedral optimization. While the default parameters work well for most cases, they may need adjustment for specific scenarios. Future work will focus on reducing parameter dependency and developing more automated parameter determination methods to improve the method's generality.

In the future, one potential direction is to extend our method by parallelizing the intersection repair process, as the repair of each intersecting region is conducted independently. Additionally, during the geometry repair process, we could consider incorporating defeaturing functionality to eliminate features that are irrelevant to simulation, thereby creating a more comprehensive mesh repair pipeline.

Acknowledgements

The authors would like to express their sincere gratitude to the anonymous reviewers for their insightful comments and valuable suggestions that have significantly improved the quality of this manuscript. Special thanks also go to the editors for their professional handling of the review process. We are also grateful to our colleagues and peers who provided helpful discussions and feedback during the development of this work.

References

[1] M. Attene, *Indirect predicates for geometric constructions*, Computer-Aided Design, 126 (2020), p. 102856.

[2] M. Bartoň, I. Hanniel, G. Elber, and M.-S. Kim, *Precise hausdorff distance computation between polygonal meshes*, Computer Aided Geometric Design, 27 (2010), pp. 580–591.

[3] M. Botsch, ed., *Polygon Mesh Processing*, A K Peters, Natick, Mass, 2010.

[4] M. Botsch and L. Kobbelt, *A remeshing approach to multiresolution modeling*, in Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing, Nice France, July 2004, ACM, pp. 185–192.

[5] X.-X. Cheng, X.-M. Fu, C. Zhang, and S. Chai, *Practical error-bounded remeshing by adaptive refinement*, Computers & Graphics, 82 (2019), pp. 163–173.

[6] G. Cherchi, M. Livesu, R. Scateni, and M. Attene, *Fast and robust mesh arrangements us-*

ing floating-point arithmetic, ACM Transactions on Graphics, 39 (2020), pp. 1–16.

[7] G. Cherchi, F. Pellacini, M. Attene, and M. Livesu, *Interactive and robust mesh booleans*, ACM Transactions on Graphics, 41 (2022), pp. 1–14.

[8] C. Dapogny, C. Dobrzynski, and P. Frey, *Three-dimensional adaptive domain remeshing, implicit domain meshing, and applications to free and moving boundary problems*, Journal of Computational Physics, 262 (2014), pp. 358–378.

[9] T. K. Dey and W. Zhao, *Approximating the medial axis from the voronoi diagram with a convergenceguarantee*, Algorithmica, 38 (2004), pp. 179–200.

[10] L. Diazzi and M. Attene, *Convex polyhedral meshing for robust solid modeling*, Sept. 2021.

[11] L. Diazzi, D. Panozzo, A. Vaxman, and M. Attene, *Constrained delaunay tetrahedrization: A robust and practical approach*, ACM Transactions on Graphics, 42 (2023), pp. 1–15.

[12] M. Dunyach, D. Vanderhaeghe, L. Barthe, and M. Botsch, *Adaptive Remeshing for Real-Time Mesh Deformation*, May 2013.

[13] J. Guo, F. Ding, X. Jia, and D.-M. Yan, *Automatic and high-quality surface mesh generation for cad models*, Computer-Aided Design, 109 (2019), pp. 49–59.

[14] J.-P. Guo and X.-M. Fu, *Exact and efficient intersection resolution for mesh arrangements*, ACM Transactions on Graphics, 43 (2024), pp. 1–14.

[15] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, *Mesh optimization*, in Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, Anaheim CA, Sept. 1993, ACM, pp. 19–26.

[16] K. Hu, D.-M. Yan, D. Bommes, P. Alliez, and B. Benes, *Error-bounded and feature preserving surface remeshing with minimal angle improvement*, IEEE Transactions on Visualization and Computer Graphics, 23 (2017), pp. 2560–2573.

[17] Y. Hu, T. Schneider, B. Wang, D. Zorin, and D. Panozzo, *Fast tetrahedral meshing in the wild*, ACM Transactions on Graphics, 39 (2020).

[18] Y. Hu, Q. Zhou, X. Gao, A. Jacobson, D. Zorin, and D. Panozzo, *Tetrahedral meshing in the wild*, ACM Transactions on Graphics, 37 (2018), pp. 1–14.

[19] P. M. Knupp, C. D. Ernst, D. C. Thompson, C. J. Stimpson, and P. P. Pebay, *The verdict geometric quality library.*, (2006).

[20] A. Lagae and P. Dutre, *Accelerating ray tracing using constrained tetrahedralizations*, in 2008 IEEE Symposium on Interactive Ray Tracing, Aug. 2008, pp. 184–184.

[21] C. Mccoid and M. J. Gander, *A provably robust algorithm for triangle-triangle intersections in floating-point arithmetic*, ACM Transactions on Mathematical Software, 48 (2022), pp. 1–30.

[22] S. Pion and A. Fabri, *A generic lazy evaluation scheme for exact geometric computations*, Science of Computer Programming, 76 (2011), pp. 307–323.

[23] C. Portaneri, M. Rouxel-Labbé, M. Hemmer, D. Cohen-Steiner, and P. Alliez, *Alpha wrapping with an offset*, ACM Transactions on Graphics, 41 (2022), pp. 1–22.

[24] H. Si, *Tetgen, a delaunay-based quality tetrahedral mesh generator*, ACM Transactions on Mathematical Software, 41 (2015), pp. 1–36.

[25] V. Skorkovska, I. Kolingerova, and B. Benes, *A simple and robust approach to computation of meshes intersection*, VISIGRAPP (1: GRAPP), (2018).

[26] Y. Wang, D.-M. Yan, X. Liu, C. Tang, J. Guo, X. Zhang, and P. Wonka, *Isotropic surface remeshing without large and small angles*, IEEE Transactions on Visualization and Computer Graphics, 25 (2019), pp. 2430–2442.

[27] Z. Xiao, J. Chen, Y. Zheng, J. Zheng, and D. Wang, *Booleans of triangulated solids by a boundary conforming tetrahedral mesh generation approach*, Computers & Graphics, 59 (2016), pp. 13–27.

[28] W.-X. Zhang, Q. Wang, J.-P. Guo, S. Chai, L. Liu, and X.-M. Fu, *Constrained remeshing using evolutionary vertex optimization*, Computer Graphics Forum, 41 (2022), pp. 237–247.

[29] Q. Zhou and A. Jacobson, *Thingi10k: A dataset of 10, 000 3d-printing models*, ArXiv, (2016).

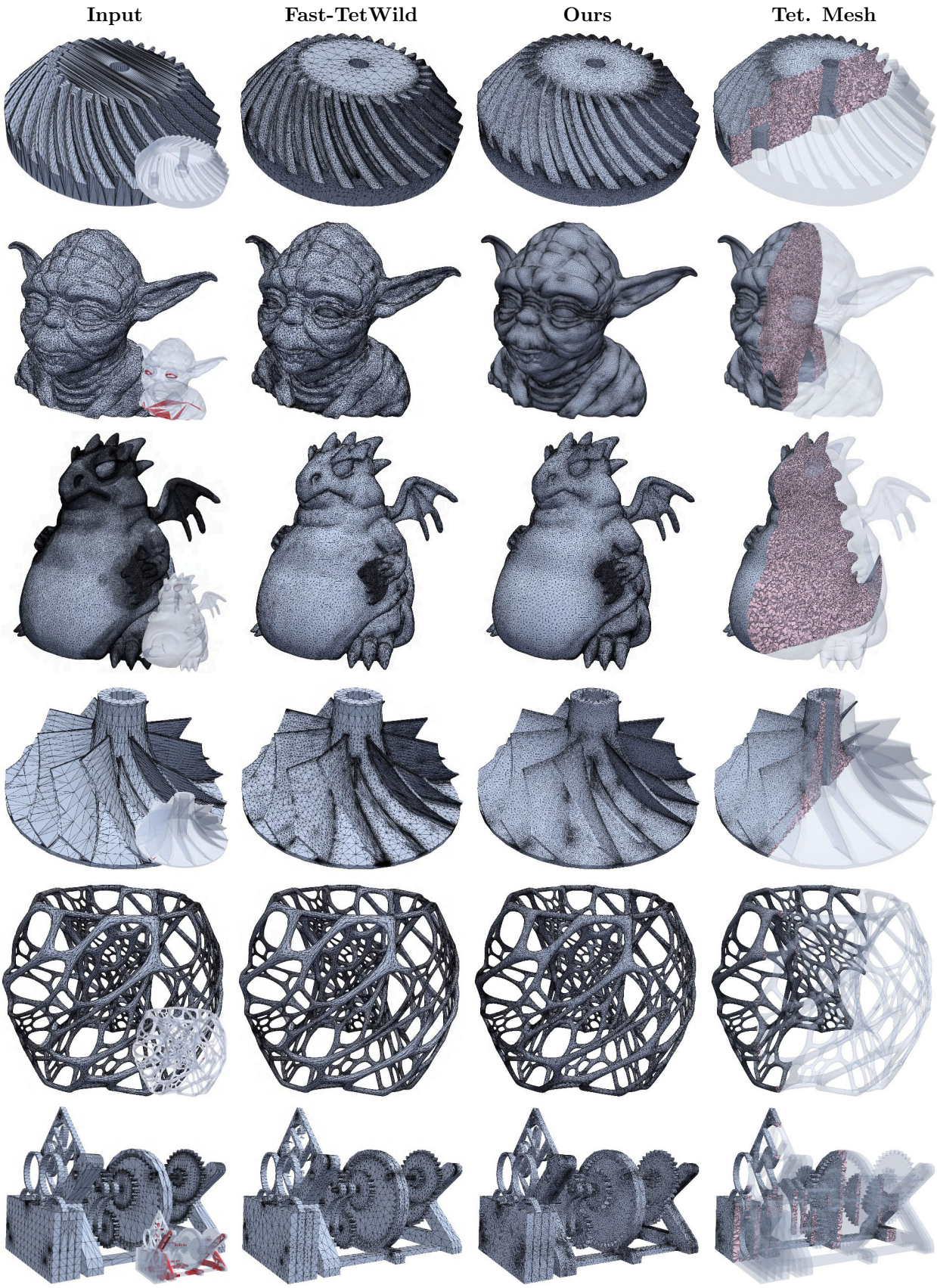


Figure 19: Mesh processing comparison on Thingi10K models. From left: input with self-intersections (red), Fast-TetWild results, our results, and tetrahedral meshes generated from our output.