Feature-Preserving Offset Meshing

HONGYI CAO, School of Computer Science, Hangzhou Dianzi University, Hangzhou, China
GANG XU, School of Computer Science, Hangzhou Dianzi University, Hangzhou, China
RENSHU GU, School of Computer Science, Hangzhou Dianzi University, Hangzhou, China
JINLAN XU, School of Computer Science, Hangzhou Dianzi University, Hangzhou, China
RABCZUK TIMON, Bauhaus University Weimar Institute of Structural Mechanics, Weimar, Germany
XIAOYU ZHANG, Beijing Institute of Spacecraft System Engineering, Beijing, China
YUZHE LUO, Zhejiang University, Hangzhou, China and LightSpeed Studios, Tencent, Bellevue, United States
XIFENG GAO, LightSpeed Studios, Tencent, Bellevue, United States

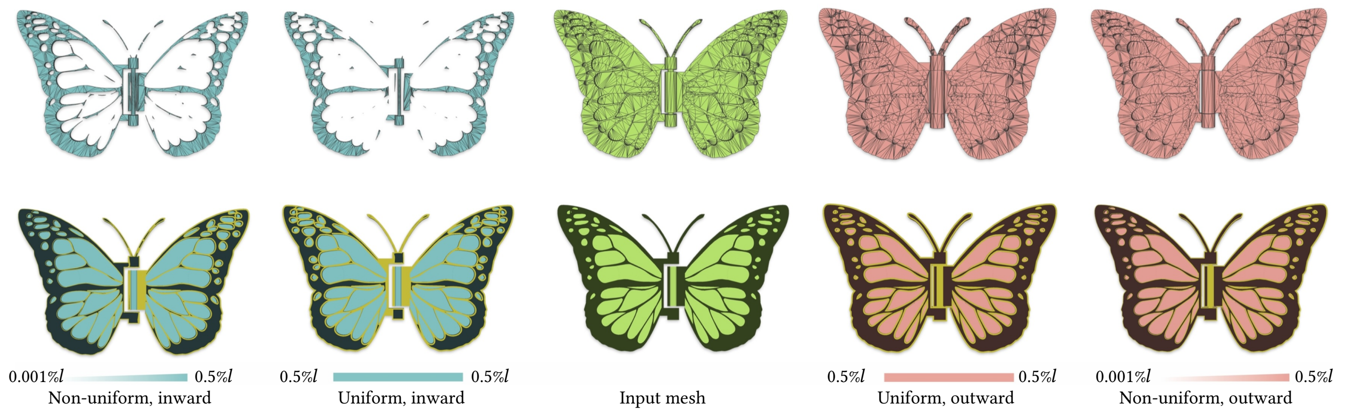


Fig. 1. Our method can generate both inward and outward offset surfaces for 3D meshes, supporting either user-specified uniform or spatially varying offset distances. For non-uniform cases, per-face distances can be arbitrarily assigned; here, we interpolate between 0.001% and 0.5% of the bounding box diagonal length. Visualizations in the accompanying figure employ the following color convention: green depicts the input mesh, red denotes outwardly offset surfaces, and blue indicates inwardly offset surfaces. The volumetric region between these offset meshes and the input—representing a solid domain—is visualized in olive. Top-row panels display full mesh views, while bottom-row cross-sectional cut views explicitly reveal this olive region through dissected planes.

We introduce a new offset meshing method that handles clean 3D surface meshes of arbitrary geometry and topology—where "clean" refers to meshes that are watertight, manifold, and free of self-intersections. Our approach also extends to imperfect, or "dirty," meshes that violate these conditions, although the problem becomes significantly more difficult in such scenarios, and faithful feature preservation near defective areas cannot always be assured. In contrast to prior techniques, which have largely focused on constant-radius offsets, our method is, to our knowledge, the first to support mitered offsets while effectively preserving sharp features. Our method is designed based on several core principles: (1) explicitly generating the offset vertices and triangles with feature-capturing energy and constraints; (2) prioritizing the generation of the offset geometry before establishing its connectivity, (3) employing exact algorithms in critical pipeline steps for robustness, balancing the use of floating-point computations for efficiency, (4) applying various conservative speed up strategies including early reject non-contributing computations to the final output. Our approach further uniquely supports variable offset distances on input surface elements, offering a wider range of practical applications compared to conventional methods.

For benchmarking purposes, we performed an extensive comparison against state-of-the-art offset methods using a curated subset of the Thingi10K dataset. Our results demonstrate the superiority of our approach over current state-of-the-art methods in terms of element count, feature preservation, and non-uniform offset distances of the resulting offset mesh surfaces, marking a significant advancement in the field.

CCS Concepts: • **Computing methodologies → Modeling and simulation;**

Additional Key Words and Phrases: Offset surface, mesh repair, variable offsets

This work is partially supported by Pioneer and Leading Goose R & D Program of Zhejiang Province (No. 2025C01086) Natural Science Foundation of China under Grant No. 62572161.
Authors' Contact Information: Hongyi Cao, School of Computer Science, Hangzhou Dianzi University, Hangzhou, Zhejiang, China; e-mail: hyc@hdu.edu.cn; Gang Xu (corresponding author), School of Computer Science, Hangzhou Dianzi University, Hangzhou, Zhejiang, China; e-mail: gxu@hdu.edu.cn; Renshu Gu, School of Computer Science, Hangzhou Dianzi University, Hangzhou, Zhejiang, China; e-mail: renshugu@hdu.edu.cn; Jinlan Xu, School of Computer Science, Hangzhou Dianzi University, Hangzhou, Zhejiang, China; e-mail: jlxu@hdu.edu.cn; Rabczuk Timon, Bauhaus University Weimar Institute of Structural Mechanics, Weimar, Thüringen, Germany; e-mail: timon.rabczuk@uni-weimar.de; Xiaoyu Zhang, Beijing Institute of Spacecraft System Engineering, Beijing, China; e-mail: xiaoyuzh.001@hotmail.com; Yuzhe Luo, Zhejiang University, Hangzhou, Zhejiang, China and LightSpeed Studios, Tencent, Bellevue, United States; e-mail: yzluo@zju.edu.cn; Xifeng Gao (corresponding author), LightSpeed Studios, Tencent, Bellevue, United States; e-mail: gxf.xisha@gmail.com.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).
ACM 0730-0301/2026/04-ART24
https://doi.org/10.1145/3797950

ACM Reference Format:
Hongyi Cao, Gang Xu, Renshu Gu, Jinlan Xu, Rabczuk Timon, Xiaoyu Zhang, Yuzhe Luo, and Xifeng Gao. 2026. Feature-Preserving Offset Meshing. *ACM Trans. Graph.* 45, 3, Article 24 (April 2026), 18 pages. https://doi.org/10.1145/3797950

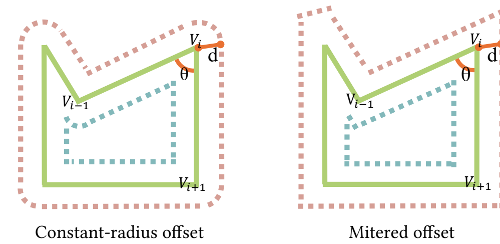


Fig. 2. Comparison of constant-radius offset (left) and mitered offset (right) for a 2D curve. In the figure, the green line corresponds to the original curve, the red line denotes the outward-offset curve, and the blue line represents the inward-offset curve.

1 Introduction

Offset mesh generation, which involves creating a parallel surface with a specific distance from a given shape, holds a place of critical importance in geometric modeling and mesh processing. This technique is fundamental in a variety of applications, including but not limited to computer-aided design and engineering, real-time rendering and animation, robotics, medical imaging, architectural design [Maekawa 1999a; Pham 1992]. For example, it is instrumental for designing mechanical parts with specific thickness requirements, such as gears and casings. In the world of animation and 3D modeling, offset surfaces enable the creation of intricate and realistic characters and environments, providing the necessary depth and complexity.

Many prior works have proposed generating offset surfaces for input meshes, such as explicit offsetting following normal directions [Jung et al. 2004], implicit distance function based iso-surfacing [Chen et al. 2023; Zint et al. 2023], Minkowski Sum based boolean approach [Hachenberger 2009], and offsetting by ray-reps based method [Chen et al. 2019b]. However, the aforementioned offset methods pertain exclusively to constant-radius offsets. In practice, offsetting techniques are broadly classified into two categories: constant-radius offset and mitered offset. While robust solutions for both types have been thoroughly established for 2D curve offsetting, with the distinctions between constant-radius offsets and mitered offsets explicitly illustrated by the example in Figure 2, constituting a well-developed body of research, their extension to 3D geometries remains conspicuously limited.

Focusing specifically on 2D mitered offsetting, a foundational approach pioneered by seminal works such as [Park and Chung 2003; Tiller and Hanson 1984; Vatti 1992] has gained wide adoption. This technique generates parallel offset edges for polygons or polylines by translating original edges along their respective normals. Crucially, it forms sharp corners (miters) at vertices instead of rounded bevels. This is achieved by computing new vertex positions through the intersection of these offset edges, while imposing a miter limit to truncate excessively elongated spikes (commonly arising at acute angles) into beveled transitions.

Notably, to the best of our knowledge, no existing methodology for 3D mesh offsetting successfully replicates the sharp vertex features and controlled spike truncation characteristic of 2D mitered offsets, establishing a significant gap in current frameworks. Moreover, the methods for constant radius offset typically suffer one or more of the following critical issues that hinder their practical usage: (1) lack of robustness in dealing with "dirty" data that may have open boundaries, non-manifold vertices and edges, and self-intersections, which is common in scanned or man-made models; (2) struggle to maintain the fidelity of the original shape, especially in handling complex geometries with sharp edges and intricate details, resulting in undesired artifacts in the offset surface; (3) computational inefficiency when the offset distance is small which can often jointly pose memory issues. These issues collectively underscore the need for an innovative solution that can address the complexities of modern geometric shapes while ensuring computational efficiency and geometry fidelity.

In this work, we propose a new explicit surface offset generation method that can address the aforementioned issues. Feature preservation is ensured by first redefining the offset distance from the traditionally employed point-to-point to point-to-plane, and then generating the offset surface geometry that can capture sharp features by solving local quadratic energies and dynamic programming. While employing the rational number representation for exact computations, we strive to be efficient by designing several acceleration strategies including parallelization through spatial domain decomposition and dynamic programming for early rejection of the expensive intersection computations.

Our solution has the additional advantages over existing approaches: (1) the resulting offset mesh is similar with the input, from both the number of triangles and connectivity aspects; (2) we support variable offset distances over different surface regions, enabling broader applications than conventional offset methods.

We empirically compare our approach against state-of-the-art on a subset of the Thingi10K dataset [Zhou and Jacobson 2016a]. Our method exhibits a significant improvement in terms of feature preservation, robustness, and element count at the same time. In addition, we provide supplementary materials that include a video and a data archive. The video is divided into two parts: the first demonstrates the offset results under incremental changes to the distance parameter, and the second presents a rotating view of the results with a 1% offset. The accompanying data archive contains the resultant models from the 1% offset operation, along with a comprehensive spreadsheet detailing the corresponding statistical metrics for each model.

2 Related Work

We briefly review prominent methods related to offset meshing, which are based on *direct offsetting*, *distance field*, *Minkowski sum*, *medial axis and skeleton*, and *ray casting*.

Direct Offsetting. A prominent method proposed by [Jung et al. 2004] offsets based on vertex normal direction. Although efficient, it struggles with holes in complex CAD models, leading to offset meshes that can be defective, particularly in scenarios involving self-intersections. The precision issues arising from floating-point operations in self-intersections have been addressed using infinite

precision operations [Campen and Kobbelt 2010], but results occasionally produce twisted meshes. Offsetting surfaces with polynomials or B-splines is explored in [Maekawa 1999b]. However, subsequent intersection operations are intricate. Another approach calculates offset positions based on distance and uniform distribution, followed by point cloud reconstruction [Meng et al. 2018].

Distance Field. These methods are popular in modern 3D printing [Brunton and Rmaileh 2021]. Generally, they require resampling to generate the final mesh, which can compromise geometric features, especially in detailed meshes [Wang and Chen 2013]. Challenges also arise from the grid density, preservation of sharp features, and computational efficiency. Some attempts, such as [Kobbelt et al. 2001], have tackled the ambiguity of the Dual Contouring and Marching Cube, but they tend to excel mostly with CAD models. In the study by [Liu and Wang 2010], there are some improvements in computational efficiency. The method introduced in [Pavić and Kobbelt 2008] tries to preserve sharp features, but it involves high grid density and computational expenses. An Octree-based method is proposed in the study by [Zint et al. 2023], which requires the input to be degenerate-free and may output offsets with self-intersections. While the feature-preserving approach introduced in [Chen et al. 2023] ensures nice geometry and topology properties of iso-surfaces, it cannot handle offset distances smaller than the grid size which is a common shortcoming shared all distance field based offset extraction methods. Among the extensive literature on offset generation, we note that only two works—[Chen et al. 2019a] and [Zint et al. 2025]—attempt to address the non-uniform offset problem. However, [Chen et al. 2019a]'s approach using dual-contouring [Ju et al. 2002] requires an impractically high grid resolution in regions with small offset distances to achieve acceptable reconstruction quality. More importantly, due to insufficient details on how interpolation is performed for varying offset distances within a grid cell, we infer that the method either incurs high computational cost for accurate in/out sign determination, or suffers from discontinuities that lead to bumpy artifacts. [Zint et al. 2025] employs a marching tetrahedra-based approach [Guéziec and Hummel 2002] to achieve variable-distance offsets with impressive results, their objective differs from ours, as our work focuses specifically on realizing a mitered offset.

Minkowski Sum Method. The Minkowski sum offers a solution for mesh offsetting by calculating the sum of mesh and sphere polygons [Rossignac and Requicha 1986]. An advanced method in [Hachenberger 2009] computes the exact 3D Minkowski sum of non-convex polyhedra by decomposing them into convex parts. Despite its robustness [CGAL v5.6], the method is slow for complex CAD models and struggles with variable thickness offsets and sharp feature preservation.

Skeleton-based Methods. Skeletal meshes are prevalent in geometry processing [Tagliasacchi et al. 2016]. The mesh model can be represented by medial axes and spheres [Amenta et al. 2001; Sun et al. 2015]. There are techniques to generate skeleton meshes [Lam et al. 1992; Li et al. 2015]. However, they usually cannot handle general models with open boundaries, self-intersections, and so on. There have been efforts to simplify medial axes [Sun et al. 2013]. Offsetting can be achieved by adjusting the radius of the balls on the skeleton. Still, these methods often fall short with models with sharp features.

Ray-based Methods. This approach [Chen et al. 2019b; Wang and Manocha 2013] is rooted in the dexel buffer structure [Van Hook 1986]. A recent algorithm in [Chen et al. 2019b] offers an efficient parallelized method using rays and voxels to compute mesh offsets, bypassing the need for distance field computation. Nonetheless, due to the grid's voxel-like structure, a high density is essential for accurate shape representation, leading to increased computational costs and dense mesh outputs. The approach also mainly considers a consistent offset, further steps have to be designed for the mesh representation conversion.

3 Method

In this section, we provide our problem statement, give the pipeline overview, explain each step in detail, and introduce the performance improvement strategies.

Problem Statement: The input of our approach contains a 3D triangle mesh M_{I} that is watertight, manifold, and free of self-intersections, an integer variable $s \in \{-1, 1\}$ to indicate the offsetting direction, and an offset scalar d_i per T_i of M_{I} where d_i could vary for different triangles. While locally, to a triangle, $s = 1$ means the offset direction has a positive product with the triangle's normal and $s = -1$ is for the opposite direction, globally we employ the generalized winding number [Jacobson et al. 2013] for inward and outward checks to robustly handle M_{I} with possibly gaps, duplicated elements, and so on.

Our goal is to generate an offset mesh M_{O} that satisfies three qualitative requirements. First, M_{O} should have the *offset distance* to M_{I} satisfying user desired value. Second, M_{O} needs to have nice geometry (no self-intersection and degeneration) to enable the easy design of automatic downstream algorithms. Third, M_{O} should contain as few as possible elements such as triangles, allowing efficient computations for the various applications of offset. We further require M_{O} to capture the geometry feature of M_{I} as much as possible, especially for prominent sharp features since we aim for reproducing the mitered offset effect.

Mitered Offset. In 2D, refer to Figure 2, the mitered offset operation on a polygon or polyline $P = \{V_0, V_1, \ldots, V_n\}$ with signed distance d translates edges along their normals to form new lines. For each vertex V_i, the miter point O_i is determined by the intersection of offset lines from adjacent edges $V_{i-1}V_i$ and V_iV_{i+1}. This is computed using their normalized offset-direction vectors $\vec{n}_1$ and $\vec{n}_2$ as

$$O_i = V_i + d \cdot \frac{\vec{n}_1 + \vec{n}_2}{1 + \vec{n}_1 \cdot \vec{n}_2}, \tag{1}$$

where $\vec{n}_1 \cdot \vec{n}_2 = \cos\phi$ with $\phi = 180^\circ - \theta$ (θ = internal angle). Valid mitering is only possible when $\theta < 180^\circ$. The miter length $M = \|O_i - V_i\|$ follows $M = \frac{d}{\sin(\theta/2)}$, with the ratio $R = \frac{M}{d}$ dictating feasibility: when R exceeds a threshold (also known as miter limit; e.g., 2.0), the junction degenerates to a bevel or fillet. This mechanism is frequently employed in industrial software and 2D drafting, such as QGIS [2024]. Extending to 3D meshes, where vertices connect to N adjacent triangular facets $\{T_1, \ldots, T_N\}$, offset planes are generated by translating each facet along its unit normal $\vec{n}_j$ by distance d. The

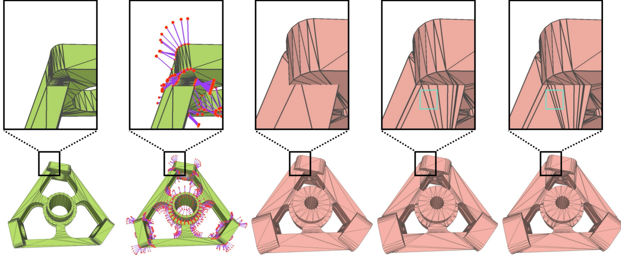


Fig. 3. Pipeline : starting from a triangle mesh M_I (left most), our approach first generates its vertices' offset points (second to the left), then builds an offset polyhedron for each of its triangle, edge and vertex (middle). After that, we convert the polyhedra set to an intersection-free triangle soup M_C by filtering out those triangles not part of M_O (second to the right), Along with some acceleration measures, this approach involves selecting only a subset of triangles from the polyhedra set for computation, which may result in the creation of some holes. Finally, by detecting the boundaries of the holes, we retrieve the triangles that were excluded in the previous step, Then we construct the connectivity of M_O (right).

ideal miter point O_i of V_i satisfies:

$$(O_i - V_i) \cdot \vec{n}_j = d \quad \forall j \in \{1, \dots, N\}. \tag{2}$$

This immediately reveals fundamental geometric limitations:

(*i*) For $N \geq 4$ or complex normal configurations, the system tends to become inconsistent, often leaving no viable solution.

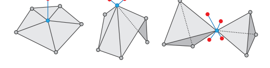
(*ii*) In cases of sharp internal angles or non-convergent normals, the offset planes diverge wildly, causing the potential miter point to recede to infinity or become undefined.

These constraints make exact geometric intersection often intractable for general meshes. Motivated by the objective of developing a 3D mitered offset capability that can accommodate facet-specific offset distances across distinct mesh facets while automatically degenerating regions exceeding the miter threshold into beveled edges—analogous to the behavior of 2D mitered offsets—we propose an optimization-based energy formulation. Importantly, this approach addresses the fundamental challenge that a simple scalar value cannot adequately define the limit condition within the complex 3D geometric context.

Method Overview: As illustrated in Figure 3, we tackle the surface offset problem by first offsetting each vertex of M_I to one or more corresponding points through the solving of a point-to-plane constrained quadratic energy (Section 3.1, Figure 3 second to the left), and then constructing a polyhedron representing the local offset volume corresponding to each vertex, edge, and triangle of M_I (Section 3.2, Figure 3 middle). Using a set of acceleration strategies, we then perform Boolean operations of the offset volumes to obtain a triangle soup, M_C, representing the geometry of M_O (Section 3.3, Figure 3 second to the right), which are finally connected to complete the generation of user desired M_O (Section 3.4, Figure 3 right).

3.1 Vertex Offset

This step is to generate the offset points for each vertex $V_i \in M_\text{I}$. As illustrated in the inset, V_i might correspond to one or more points depending on different local configurations, such as non-manifoldness, saddle region, and so on. For easy explanation, we first introduce the simple but effective linear constrained quadratic optimization solution when V_i has only one offset point, and then describe how to tackle the general scenario, i.e., multiple offset points, via dynamic programming.



A vertex (blue) of M_I can have one or more offset points (red).

Linearly Constrained Quadratic Optimization. Let O_i be V_i's unique offset, we assume the local region of V_i is formed by a triangle set $\mathcal{T}_i$ where every triangle $T_j \in \mathcal{T}_i$ is equipped with an offset distance d_j. Denote $N = |\mathcal{T}_i|$. Note that, since we assume the input could have an arbitrary topology, we set $\mathcal{T}_i$ to be the set of triangles that V_i is directly adjacent to if the boundary of this triangle set contains a simple circle topology. Otherwise, we consider $\mathcal{T}_i$ as the set of all triangles with an ϵ distance from V_i. We set $\epsilon = 10^{-5} l$ by default, where l is the length of the diagonal of M_I's bounding box.

By requiring the point-to-plane distance from O_i to $T_j \in \mathcal{T}_i$ to be d_j, we can compute O_i by solving the following optimization:

$$\underset{O_i}{\text{argmin}} \; ||O_i - V_i||^2 \quad \text{s.t. } \text{P}_j^\text{T} O_i = d_j \quad \forall j < N, \tag{3}$$

where $\text{P}_j = [a\, b\, c\, d]^\text{T}$ represents the plane of triangle T_j defined by the equation $ax + by + cz + d = 0$ and $a^2 + b^2 + c^2 = 1$. The optimization term $||O_i - V_i||^2$ is introduced to pick the position of O_i nearest to V_i when the solution space is a plane or a line. While Equation (3) may be solved using OSQP solver [Stellato et al. 2020], due to numerical precision issues, we solve it as a two step process. We first solve an unconstrained quadratic problem:

$$\underset{O_i}{\text{argmin}} \; \lambda ||O_i - V_i||^2 + \sum_j \left(\text{P}_j{}^\text{T} O_i - d_j\right)^2 \quad \forall j < N. \tag{4}$$

The role of λ is to ensure that, in cases where there are infinitely many solutions satisfying the point-to-plane distances, an optimal solution can be selected, while minimizing its impact on other scenarios. Consequently, it is necessary to set it to a very small number. By default, we set λ to 10^{-9}. Subsequently, this expression can be solved using the least squares method. We employ the QR decomposition from the Eigen library for the solution.

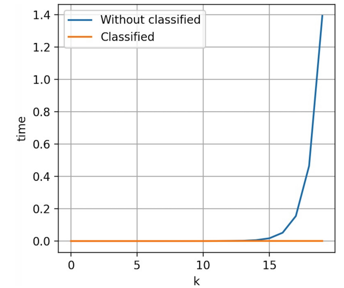


Fig. 4. The two curves respectively demonstrate the computational time (in seconds) required to calculate a point with and without prior classification. It can be observed that when K is particularly large, lacking this classification makes it difficult to compute the offset points within a reasonable timeframe.

We then perform a distance check for each plane to ensure $|\text{P}_j^\text{T} O_i - d_j| \leq \alpha$, where α is a parameter to control how much the desired point-to-plane distance of the user is satisfied. Due to the presence of λ, the distances between the solved point position and the planes will always be slightly off from the desired values. Therefore, α cannot be set to zero, but must be assigned a small value. We set $\alpha = 10^{-6}$ by default.

To handle cases where the distance check may not be satisfied for all triangles in $\mathcal{T}_i$, we propose generating multiple offsets for a vertex, as elaborated in the following section. Although Jiang et al. [2020] adopted a quadratic programming-based strategy for offset vertex generation that shares certain similarities with our approach, they encountered challenges in solving the optimization and resorted to the use of singular points—a strategy that proves inadequate in our context. Therefore, we introduce the following method to address this limitation.

Dynamic Programming. When there is no solution found for Equation (5), we compute multiple offset points. We propose to solve this issue by dividing the triangle set $\mathcal{T}_i$ into K subsets, denoted by $\mathcal{Q} = \{\mathcal{T}_i^0, \mathcal{T}_i^1, \ldots\}$ where $|\mathcal{Q}| = K$, and we compute an offset point O_i^k for each subset $\mathcal{T}_i^k$. Denote $\mathcal{R}$ as the offset point set, i.e., $\{O_i^0, O_i^1, \ldots\}$ and $|\mathcal{R}| = K$. Therefore, we need to solve the following problem:

$$\underset{\mathcal{Q},\mathcal{R}}{\operatorname{argmin}} \ \lambda \sum_k \|O_i^k - V_i\|^2 + \sum_k \left(\mathrm{P}_j^{k\mathrm{T}} O_i^k - d_j^k\right)^2 \tag{5}$$

The challenge arises for deciding the K subsets while ensuring the point-to-plane distance constraint for the planes of $\mathcal{T}_i^k$ for each O_i^k. While we can simply decompose $\mathcal{T}_i$ based on the similarity of the normal of the triangles, solution of O_i^k is still not guaranteed to exist and normal could be wrongly computed when floating-point precision is involved. Instead, we propose a dynamic programming strategy to solve $\mathcal{Q}, \mathcal{R}$ as detailed below, where Algorithm 1 is the pseudo-code. Given the N triangles in $\mathcal{T}_i$, a subset could have 2^N possible configurations. We encode each subset using the binary representation with length N. For example, $0b010101$ represents the subset of $\{T_0, T_2, T_4\}$ for $N = 6$, while $0b100100$ is the subset of $\{T_2, T_5\}$. We use D_o to store the energy values of Equation (3) when solving an offset vertex O_i^k for the group of triangles $\mathcal{T}_i^k$ represented in the aforementioned binary format x, and D_c to store the coordinates of the corresponding offset vertex O_i^k. As shown in Algorithm 1, we employ a recursive scheme (lines 2-12) to find the best group decomposition, where for each level of the recursion we decompose a set x into two subsets $\tilde{x}$ and $x - \tilde{x}$ if x cannot be solved by Equation (3). Here, the union of the triangles represented by $\tilde{x}$ and $x - \tilde{x}$ is the same as the triangle set of x. In this case, we use the an array of scalars, D_s, to record the best decomposition for x, i.e., $D_s[x] = \tilde{x}$. From which, we can retrieve all the decomposed subsets of $\mathcal{T}_i$, as shown by lines 15-20 of Algorithm 1. Note that, the dimension of D_o, D_c, and D_s are all $2^N - 1$.

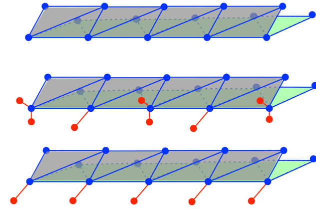


Fig. 5. The upper section of this figure depicts a local region of the mesh that requires outward offset. The middle section illustrates the result obtained without considering neighboring vertices during the solution process, while the lower section shows the outcome when neighborhood information is incorporated. The angle formed by the gray face and the green face admits two solutions, which are nearly indistinguishable in our defined energy function.

The introduced dynamic programming has a computational complexity of $O(3^N + 2(1 - 2^N))$ [A10 2024; Giraudo 2015]. The computation is efficient for $N \leq 12$, e.g., $\leq 0.01s$ without any parallelization. Since the dynamic programming operates independently in each local context, it can be fully parallelized. However, if N becomes excessively large, the computation time may grow significantly as show in Figure 4. To address this problem, we first classify the planes defined by $\mathcal{T}_i$ into groups where the planes within a group have the similar normal. We then compute an average plane for each group and consider the averaged one as the plane for all the triangles with each group. To summarize, we compute the offsets for each vertex of M_I by firstly merging its adjacent triangles with similar normals, and then perform Algorithm 1 to solve the offset points. Figure 7 illustrates different offset cases on a real example.

Our implementation also incorporates a practical optimization to reduce local wrinkling artifacts caused by numerical instabilities. Specifically, when processing sharp geometric features within the mesh, multiple grouping schemes may exhibit computational indistinguishability in achieving minimal energy states per Equation (5). To resolve this, our algorithm processes vertices in parallel via a multi-threaded **breadth-first search** (**BFS**) traversal. For each vertex v, we store all candidate grouping solutions whose energy lies within a specified tolerance threshold (nominally set to 3% in our experiments gained from experience) of the lowest-energy solution identified for v. The solution selection for v proceeds as follows: (1) With processed neighbors: If adjacent vertices have already been processed, we select the candidate solution for v whose offset direction vector most closely aligns with the grouping solutions present in the neighboring vertices. (2) Without processed neighbors: If no adjacent vertices have been processed, we directly select the candidate solution for v with the minimum energy. A schematic diagram of the aforementioned optimization is presented in Figure 5. Furthermore, Degenerate triangles, which lack normal vectors, do not satisfy our equation. To address this, we introduce a preprocessing step before the main process.For the detection and handling of degenerate triangles, all computations are performed using rational numbers with infinite precision. The detection of degenerate triangles is made unequivocal through the use of rational numbers, which allows us to identify all degenerate triangles without ambiguity. These triangles are classified into two cases: (1) a triangle where one of its edges has a length exactly equal to zero, and (2) a triangle where no edge has a length equal to zero, but the sum of the lengths of two edges is exactly equal to the length of the third edge. The first case is relatively simple to handle; we remove the degenerate triangle and merge the two vertices at the endpoints of the zero-length edge. In the second case, vertex merging alone is insufficient, and new edges must be introduced to split the affected triangles. The process is as follows: First, all degenerate triangles of the second type are removed by deleting only the faces while retaining the vertices. Next, we iterate over all edges and identify any additional vertices lying along the line segment represented

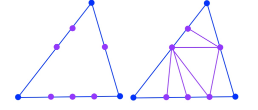


Fig. 6. This figure illustrates the subdivision strategy for resolving degenerate triangles. The left side depicts the input face before subdivision, where blue lines represent the three edges of the triangle and purple points denote the additional vertices identified as lying precisely on each edge. The right side presents our corresponding subdivision strategy.

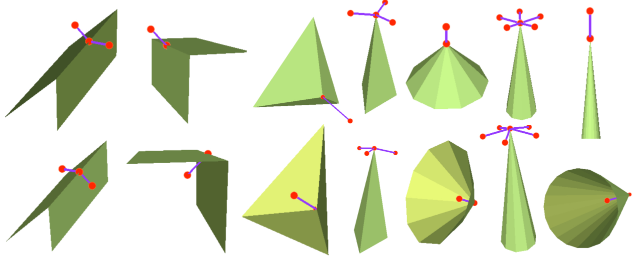


Fig. 7. Nine different vertex offset scenarios computed by our approach on a mesh example. Each column corresponds to one scenario, where the top and bottom rows show two different offset direct (outward and inward). It can be observed that in some sharp cases, the points generated by the inward offset of vertices may appear outside the input model. This situation does not pose any issues. The subsequent convex hull and winding number computations will exclude these points.

ALGORITHM 1: Dynamic Programming for Vertex Offset

```
    Input: V_i, T_i, d, T_j ∈ T_i
    Output: Q, R
 1: Set each entry of D_o, D_c, and D_s as NULL
 2: Function DFS(x):
 3:   | if D_o[x] ≠ NULL then
 4:   |   | return D_o[x]
 5:   | D_o[x] ← ∞
 6:   | if Equation (3)(x) is solvable then
 7:   |   | Set Equation (3)(x)'s energy and solution to D_o[x] and
      |   |   D_c[x]
 8:   | for each x̃ derived from x do
 9:   |   | o ← DFS(x̃) + DFS(x − x̃)
10:   |   | if o < D_o[x] then
11:   |   |   | D_o[x] ← o, D_s[x] ← x̃
12:   | return D_o[x]
13: DFS(2^N − 1)
14: Q̃ ← ∅, R ← ∅, Q̃.push(2^N − 1)
15: while Q̃ ≠ ∅ do
16:   | x ← Q̃.front(), Q̃.pop()
17:   | if D_s[x] = NULL then
18:   |   | Q.insert(x), R.insert(D_c[x])
19:   | else
20:   |   | Q̃.push(D_s[x]), Q̃.push(x − D_s[x])
```

by each edge (excluding the endpoints). These vertices are used to divide the edge into smaller segments. Subsequently, vertices that occupy the same spatial position are merged. Finally, we traverse all faces, and based on the split status of the three edges associated with each face, we subdivide the face into smaller triangles. The subdivision strategy is illustrated in Figure 6.

3.2 Local Offset Polyhedron

After generating the offset points of M_I, this step is to generate the local offset volumes for the elements of M_I, i.e., vertices, edges, and triangles. Ideally, if there is only one offset point for each vertex, then we only need to generate the local offset volume P_i for each triangle, $T_i \in M_\mathrm{I}$, and P_i is simply the triangle prism formed by T_i's three vertices and their corresponding offset points as illustrated in Figure 8 left.

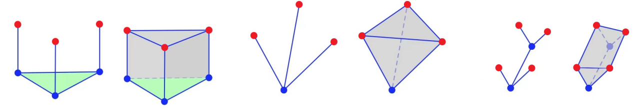
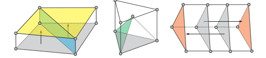


Fig. 8. From left to right, the diagrams respectively illustrate the construction of offset volumes from a triangle, a vertex, and an edge.

In practice, however, one vertex may have multiple offset points, the offset volume computation for M_I would be slightly involved. We tackle the problem by computing an offset polyhedron for each vertex, edge, and triangle of M_I, respectively. For each element, its offset polyhedron is computed as the convex hull of a point set associated with the element. To be specific, for a vertex V_i of M_I, the point set includes V_i and all its offset points $\{O_i^0, O_i^1, \cdots\}$ corresponding to the neighboring triangle set $\mathcal{T}_i$ (Figure 8 middle). For an edge $E_i = \{V_0, V_1\}$ of M_I, its polyhedron is computed as the following (Figure 8 right). Assume E_i is adjacent to a set of triangles $\mathcal{T}_{E_i}$, the point set of E_i for the convex hull computation includes V_0, V_1, and those from V_0's and V_1's offset points satisfying the following condition: the offset point's corresponding triangle subset solved by Algorithm 1 contains any triangle of $\mathcal{T}_{E_i}$. Similarly, for a triangle $T_i = \{V_0, V_1, V_2\}$ of M_I, its polyhedron is the convex hull of the point set including V_0, V_1, V_2, and those offset points of V_0, V_1, V_2 as long as the offset point's corresponding triangle subset solved by Algorithm 1 contains T_i.

This strategy leads to a set of polyhedra, all comprising exclusively triangular faces, designated as $\mathcal{P}$.

3.3 Geometry Extraction



A face of M_C is with one of the illustrated five types. Left: $\mathrm{T_{II}}$ (blue), $\mathrm{T_{IV}}$ (gray), $\mathrm{T_V}$(yellow); middle: $\mathrm{T_I}$ (green); right: $\mathrm{T_{III}}$ (red).

After generating the offset polyhedra, this section is to compute all the triangles that constitute the geometry of M_O in three steps. First, as intersections occur between polyhedra in $\mathcal{P}$, we convert all triangles from $\mathcal{P}$ into a conforming triangle mesh M_C by resolving all the intersections, where the intersection points and line segments are taken as constraints for a subsequent Delaunay triangulation for each face of $P_i \in \mathcal{P}$. After this step, M_C is intersection-free and contains only triangles. As illustrated in the inset, a triangle of M_C can be classified into one of the five types, i.e., enclosed by a P_i ($\mathrm{T_I}$), shared by two P_is that are at different sides of the triangle ($\mathrm{T_{II}}$), has an opposite sign to the offset direction as evaluated by M_I ($\mathrm{T_{III}}$), a triangle of M_I ($\mathrm{T_{IV}}$), and a face of M_O ($\mathrm{T_V}$). Since $\mathrm{T_{IV}}$ faces are properly tracked from the beginning, our next two steps are to obtain the triangles with type $\mathrm{T_V}$ by filtering out those triangles belonging to the first three types.

Second, we identify and discard triangles with $\mathrm{T_I}$ and $\mathrm{T_{II}}$ through rational number represented ray intersection checks where the rays are shot from the centers of these faces along their normal directions. Both types can be easily detected by checking if there are any intersections between the rays and other polyhedra other than the ones these faces belong to.

Third, we detect faces with T_{III} through the generalized winding number [Jacobson et al. 2013]. Since the winding number computation is not exact, when the offset distances d are close to the numerical precision, T_{III} faces may be wrongly labeled, resulting redundant faces in M_O. However, accordingly to our extensive experiments, we don't observe such an issue for $d \geq 10^{-6}$ for both uniform and varying offsets. Note that, if the input mesh is watertight and free of self-intersections, T_{III} faces can be filtered out completely by employing [CGAL v5.6].

The binary classification of spatial regions as interior or exterior differs fundamentally for watertight versus non-watertight meshes. For geometrically closed surfaces, meaning watertight surfaces, ray intersection tests using parity counting provide unambiguous inside/outside determinations. This is mathematically guaranteed by the Jordan-Brouwer separation theorem [Alexander 1922]. In contrast, non-watertight meshes violate topological closure, rendering such deterministic approaches inapplicable and necessitating heuristic definitions that balance physical plausibility against computational tractability. The winding number criterion has emerged as the predominant heuristic, which has emerged as a widely-adopted methodology for this purpose, particularly exemplified in tetrahedral meshing frameworks such as TetWild [Hu et al. 2018] where its computational efficiency and mathematical rigor prove advantageous. While this approach effectively resolves paradoxical configurations where inward offsets extrude beyond the original mesh boundary or outward offsets intrude into ostensibly internal regions, it remains susceptible to numerical precision limitations, Given these tradeoffs, the winding number methodology persists as our selected approach due to its operational efficacy and robust in canonical cases, notwithstanding its propensity to induce localized errors under specific situation.

3.4 Topology Construction

After obtaining M_C, i.e., the triangle soup composing the geometry of M_O, in this section, we accomplish the connectivity of M_O to make it free of degenerate elements and intersections under exact-number representations. M_O is also watertight given that the input is watertight and self-intersection-free.

We execute three steps to convert M_C to M_O. First, we merge the vertices with the same coordinates in rational numbers so that all triangles are connected but zero-area holes may still exist. Zero-area holes occur because our intersection testing intentionally skips detection of purely edge-edge intersections between triangles. While this design choice substantially accelerates computation, it can create topological anomalies at these co-incident edges, which require further processing.

Second, we perform a hole-filling to remove the zero-area gaps. Given a boundary loop, our hole-filling is done by inserting an edge at a time with two criteria: this edge has no intersection with any other triangles and the areas of the newly generated polygons are still zero, until the loop is fully triangulated. Third, the newly generated degenerate triangles are categorized into the two types mentioned in Section 3.1 and are addressed by their respective methods described above, thus eliminating all degenerate elements introduced during the hole-filling step without altering the geometry of the M_O.

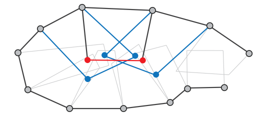
3.5 Speedup Strategies

We employ speedup strategies from different aspects to improve the performance of our approach as detailed below.

We use OpenMP to trivially parallelize the vertex offset generation (Section 3.1) and the triangle polyhedra construction (Section 3.2). Furthermore, we spatially partition the spatial domain, which is covered by the geometry formed by extending M_I along three axes with the largest offset distance, into cubes. Each cube is associated with the list of triangles of M_I that are either fully contained or have intersections with the cube. So the computational domain of M_O is constrained within each cube. We apply this strategy for Sections 3.3 and 3.4.

Data structure-wise, we employ AABB tree data-structure as much as possible for spatial search of vertex, face, and polyhedron elements, such as the $\mathcal{T}_i$ collection for V_i in Section 3.1, the polyhedron neighborhood search during intersection resolving, the ray-intersection check and the winding number computation during filtering invalid triangle in Section 3.3.

Based on the performance statistics, we find that the most time-consuming part is during the intersection resolving in Section 3.3 where an extensive number of exact computations [Hachenberger et al. 2007] are involved. While all intersections among the faces of the polyhedra of $\mathcal{P}$ are resolved by splitting the faces into intersection-free triangles, many of the resulting triangles are not part of M_O. In another words, if the entire or the most region of a face will be discarded for computing M_O, then most of the computations for resolving the intersections with this face are wasted since most of the resulting triangles will be discarded as well. Based on the principle that postponing unnecessarily intersection computations as much as possible, we propose a speedup strategy of Section 3.3 with the pseudo-code given by Algorithm 2.



While many polygons intersect with the edge in red, two polygons in blue are enough to early reject the edge from being further decomposed into T_{II} edges.

Assume $\mathcal{T}$ is the set of triangles of all polyhedra of $\mathcal{P}$ and $\mathcal{T}_o$ is the set of all triangles of M_O. We first subdivide $\mathcal{T}$ into two sets $\mathcal{T}_{cur}$ and $\mathcal{T}_{later}$ where they contains triangles with high and low chances to be part of M_O, respectively. As shown in lines 2-10 of Algorithm 2, as an initialization, $\mathcal{T}_{later}$ contains triangles with two scenarios that are both unlikely to be part of M_O: sharing a vertex with M_I and is inside (outside) M_I for $s = 1$ ($s = -1$). We then obtain $\mathcal{T}_o$ in an iterative way by constructing the most of it using $\mathcal{T}_{cur}$ (lines 12-23) and gradually refining it using $\mathcal{T}_{later}$ (lines 11 and 24-29). By doing so, the output of Algorithm 2 is ensured to be same as the brute-force approach in Section 3.3. During the construction of $\mathcal{T}_o$ using $\mathcal{T}_{cur}$, we identify a scenario that for a triangle of $\mathcal{T}_{cur}$ that intersects with many polyhedra, a large part of it may be covered by one or several of those polyhedra. For such faces, there would be many intersection computations to produce T_I and T_{II} sub-triangles that are eventually discarded through ray-casting test. The inset illustrates a simplified scenario in 2D. Accordingly, in lines 17-22 of Algorithm 2, we propose to subdivide T_i into triangles by a subset of all its intersecting polyhedra to avoid unnecessarily more detailed intersection computations. Specifically, given a triangle T_i that has intersections with $N_p \geq 5$ polyhedra denoted

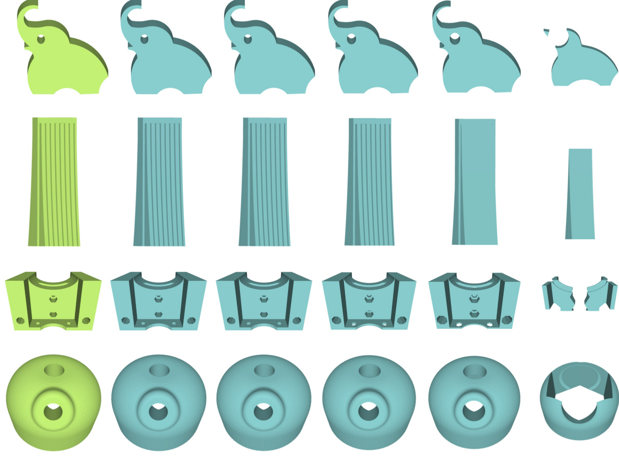


Fig. 9. This figure shows the results of our algorithm for inward offsets at different distances. From left to right, the images are the input, 0.05%l, 0.1%l, 0.5%l, 1%l, and 5%l.

ALGORITHM 2: Speed up of Section 3.3

Input: $\mathcal{T}$, $\mathcal{P}$
Output: $\mathcal{T}_o$
1: Set $\mathcal{T}_{cur}$, $\mathcal{T}_{later}$, and$\mathcal{T}_o$ as $\emptyset$
2: **for** *each* $T_i \in \mathcal{T}$ **do**
3: **if** $s = 1$ *and Winding_number(centroid(T_i))* > 0.5 **then**
4: $\mathcal{T}_{later}$.push(T_i)
5: **else if** $s = -1$ *and Winding_number(centroid(T_i))* ≤ 0.5 **then**
6: $\mathcal{T}_{later}$.push(T_i)
7: **else if** *(T_i) have a vertex on M_i* **then**
8: $\mathcal{T}_{later}$.push(T_i)
9: **else**
10: $\mathcal{T}_{cur}$.push(T_i)
11: **while** $\mathcal{T}_{cur} \neq \emptyset$ **do**
12: $\mathcal{T}'_{cur} \leftarrow \emptyset$
13: **for** $T_i \in \mathcal{T}_{cur}$ **do**
14: **if** $N_p < 5$ **then**
15: $\mathcal{T}'_{cur}$.push(T_i)
16: **else**
17: Subdivide T_i into $T_i^0, T_i^1, T_i^2, T_i^3$
18: Compute P_i^j containing T_i^j the most, $j = \{0, 1, 2, 3\}$
19: $\tilde{\mathcal{T}}_i \leftarrow$ resolving intersections between T_i and $P_i^0, P_i^1, P_i^2, P_i^3$
20: **for** $t_j \in \tilde{\mathcal{T}}_i$ **do**
21: **if** *t_j is not contained in $P_i^0 || P_i^1 || P_i^2 || P_i^3$* **then**
22: $\mathcal{T}'_{cur}$.push(t_j)
23: $\mathcal{T}_o \leftarrow$ perform Section 3.3 of $\mathcal{T}'_{cur}$ and $\mathcal{P}$
24: $\mathcal{T}_{cur} \leftarrow \emptyset$
25: $\mathcal{B} \leftarrow$ Boundary of $\mathcal{T}_o$
26: **for** $T_i \in \mathcal{T}_{later}$ **do**
27: **if** *T_i intersects with $\mathcal{B}$* **then**
28: $\mathcal{T}_{cur}$.push(T_i)
29: $\mathcal{T}_{later}$.delete(T_i)

by the set $\mathcal{P}_i$, we first subdivde T_i into four smaller triangles, i.e., T_i^j with $j = \{0, 1, 2, 3\}$, using the mid-point subdivision. For each

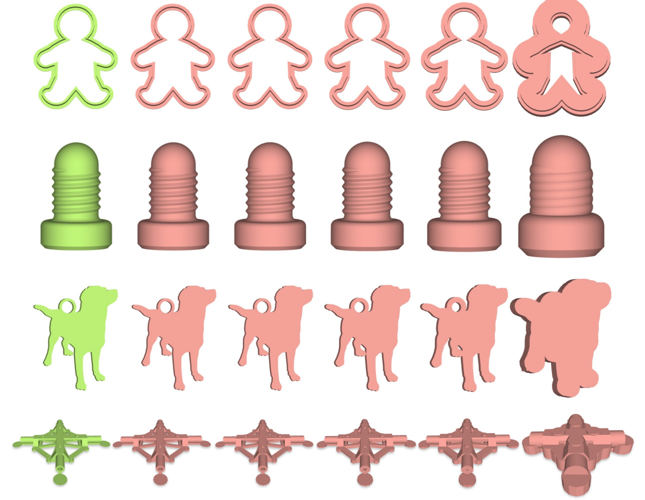


Fig. 10. This figure shows the results of our algorithm for outward offsets at different distances. From left to right, the images are the input, 0.05%l, 0.1%l, 0.5%l, 1%l, and 5%l.

T_i^j, we then find the polyhedron from $\mathcal{P}_i$ that contains T_i^j the most by creating several sampling points within T_i^j and checking the number of sampling points contained by the polyhedron. After that, corresponding to T_i^j, there will be four polyhedra identified, P_i^j with $j = \{0, 1, 2, 3\}$. We then perform the intersection resolving procedure in Section 3.3 for T_i and these four polyhedra to obtain a set of subdivided triangle set of $\tilde{\mathcal{T}}_i$. After filtering out those triangles in $\tilde{\mathcal{T}}_i$ that are fully contained in any P_i^j, $\mathcal{T}_{cur}$ will be updated.

By experimenting on several models randomly chosen from our testing dataset, we find that the early filtering strategy helps our pipeline achieves ~30x performance gain in a large offset distance e.g., more than 1%.

4 Experiments

eW run all our experiments on a desktop with a 10-core Intel processor clocked at 5.30 GHz and 32 GB of memory. We implemented our approach in C++, with CGAL [CGAL v5.6] for exact computations. Throughout the entire article, our offset distance parameter d is expressed as the ratio to l where l is the length of the diagonal of the input's bounding box, e.g., $d = 1\%$ represents that the offset distance is actually 1%l. Please refer to the supplementary material for more information.

4.1 Metrics

Mesh Distances. We employ the typically used point-to-point distance H_{point} to measure the mesh distance between M_O and M_I. Since we aim at generating mitered offset instead of the traditionally constant-radius offset, we also propose to measure point-to-plane distance H_{plane} between M_O and M_I.

Given a vertex v and a triangle mesh $\tilde{M}$, we denote $d_v(v, \tilde{M})$ as the closest Euclidean distance from v to $\tilde{M}$ and $\tilde{v} \in \tilde{M}$ is the point on $\tilde{M}$ closest to v. We further denote $d_t(v, \tilde{M})$ as the Euclidean

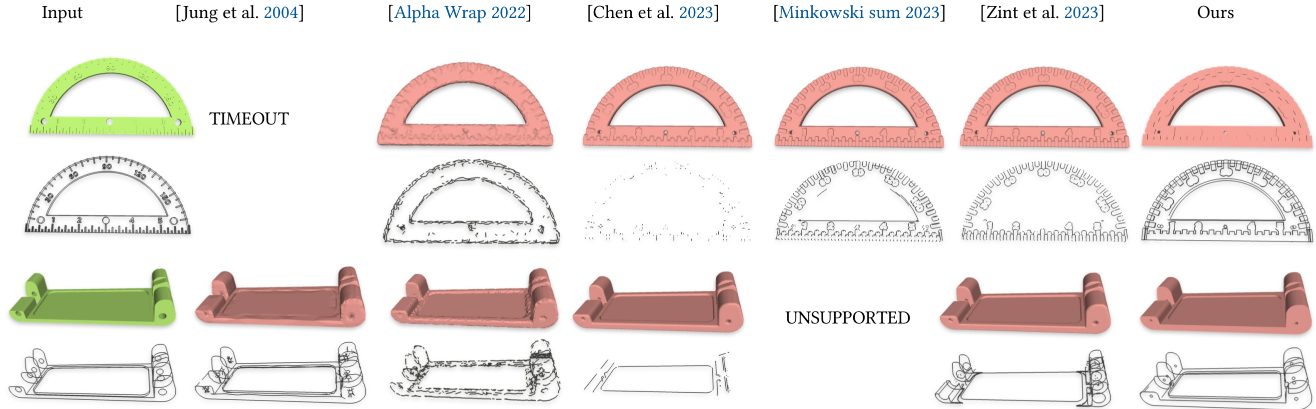


Fig. 11. This figure shows an outward offset of 1%l. The left side is the input, and the back shows the comparison results of six methods. It can be seen that our method performs better in restoring feature lines.

distance from v to the plane passing through $\tilde{v}$ and tangent to $\tilde{M}$. The computations of H_{point} and H_{plane} are summarized in Equation (6).

$$
\begin{aligned}
H_{point}(M, \tilde{M}) &\triangleq \frac{1}{|M|}\int_M d_v(v, \tilde{M})ds, \\
H_{plane}(M, \tilde{M}) &\triangleq \frac{1}{|M|}\int_M d_t(v, \tilde{M})ds,
\end{aligned} \tag{6}
$$

$H_{point}(M_{\mathrm{O}}, M_{\mathrm{I}})$ and $H_{plane}(M_{\mathrm{O}}, M_{\mathrm{I}})$ measure the average distances from M_{O} to M_{I}. The sampling strategy in Metro [Cignoni et al. 1998] is used for their computation. We use $D_{point} = |H_{point} - d|$ and $D_{plane} = |H_{plane} - d|$ to indicate how close of our generated mesh aligns with the user specified offset distance, where the smaller of D_{point} and D_{plane} the better.

Feature Preservation. Moreover, to quantitatively measure the effectiveness of various approaches in feature preservation, we propose to compute the difference between sharp features of the input and output meshes. Specifically, based on the simple angle-thresholding strategy [Gao et al. 2019], we first detect feature lines $S_{M_{\mathrm{I}}}$ and $S_{M_{\mathrm{O}}}$ of M_{I} and M_{O}, respectively (black lines shown in Figure 11 and Figure 12), then we uniformly sample $S_{M_{\mathrm{I}}}$ and $S_{M_{\mathrm{O}}}$. For each sample $v \in S_{M_{\mathrm{O}}}$, we can find a closest sample $\tilde{v} \in S_{M_{\mathrm{I}}}$. We denote $d_{angle}(v, S_{M_{\mathrm{O}}})$ as the l_2 norm of the dihedral angles of the edges where v and $\tilde{v}$ belong to. The difference between $S_{M_{\mathrm{O}}}$ and $S_{M_{\mathrm{I}}}$ can be calculated by

$$
D_{angle}(M_{\mathrm{O}}, M_{\mathrm{I}}) \triangleq \frac{1}{|S_{M_{\mathrm{O}}}|}\int_{S_{M_{\mathrm{O}}}} d_{angle}(\tilde{v}, S_{M_{\mathrm{I}}})dl, \tag{7}
$$

where $D_{angle}(M_{\mathrm{O}}, M_{\mathrm{I}})$ measure the similarity of dihedral angles from $S_{M_{\mathrm{O}}}$ to $S_{M_{\mathrm{I}}}$.

4.2 Our Results

Our method requires only a single user-specified parameter, offset distance d, and generates the corresponding offset surface mesh that has the following unique properties.

Mitered Offset. Aiming at generating the mitered offset surface, our approach provides the unique advantage over existing methods

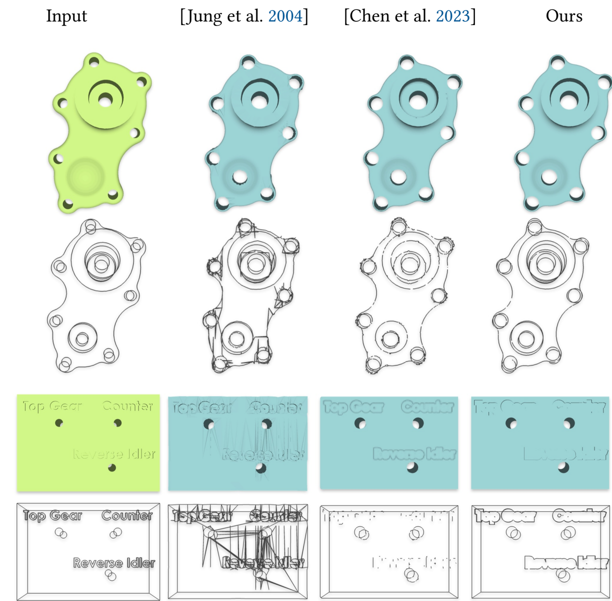


Fig. 12. This figure shows an inward offset of 1%l. The left side is the input, and the back shows the comparison results of three methods. It can be seen that our method performs better in restoring feature lines.

by respecting the features of the input excellently, as shown in Figures 9 – 12 and Figure 26.

Unsurprisingly, as demonstrated in Figure 13, as the offset distance gets larger, our generated surface tends to be a shape with a rectangular shape, distinct from the constant-radius offset with the tendency of being a sphere.

Non-uniform Offset Distance. By varying the offset distance d for different triangles of input, our approach can easily generate offset meshes with varying offset distances, as illustrated in Figure 14, while maintaining the sharp features of the original meshes. The calculation of variable-distance offsets inherently exacerbates the unsolvable possibility of the optimization formulation in Section 3.1, as divergent offset magnitudes on adjacent faces impose

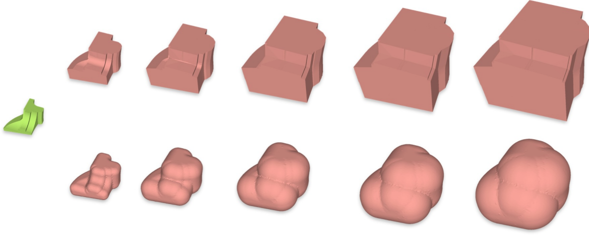


Fig. 13. Display the large offset distance. It can be observed that generating an offset with a large offset distance using the original vertices may lead to creases due to the resolution limited by the number of vertices. The offset distance from left to right is $10\%l$, $20\%l$, $30\%l$, $40\%l$, $50\%l$ with outer direct. The first row is our method result, the second row shows a radius-based method.

conflicting positional constraints on shared vertices. When numerical solvers fail to reconcile these constraints, particularly under large discontinuities in the 1-ring neighborhood, vertex splitting occurs to preserve geometric distance fidelity, with extreme cases generating a number of offspring vertices equal to the local face count. Although this mechanism ensures mathematical consistency with input distances, it manifests itself as a wrinkle that degrades visual and functional smoothness.

To elucidate this mechanism, consider a minimal example where a vertex V_i has all its 1-ring neighboring faces coplanar, yet exhibits exactly two conflicting offset distances d and $k + d$ (with $k > 0$) across these faces:

- When $k \leq 2\alpha$, the optimization merges the offsets by relocating a point to a distance of $k + d - \alpha$. Since this distance differs from both $k + d$ and d by no more than α, it simultaneously satisfies both constraints, thereby achieving a smooth transitional geometry through controlled distance relaxation.
- If $|k| > 2\alpha$, the formulation becomes infeasible, necessitating the splitting of V_i into two distinct vertices with offsets d and $k + d$, respectively. Subsequent convex hull computations between these derived vertices produce sharp, step-like features that preserve geometric discontinuities.

To mitigate this, as shown in Figure 15, the parameter α can be relaxed if end-users can tolerate minor deviations between the realized and target offset distances.

Topology. The topology of our generated mesh is similar to the input mesh, and the density of the mesh is close to or slightly greater than the input mesh as shown in Figure 33.

4.3 Dirty Meshes

Our method assumes that the input mesh is watertight, manifold, and free of self-intersections. However, it can also be applied to imperfect or "dirty" meshes that violate these conditions. In such cases, the problem becomes significantly more challenging, and faithful feature preservation near defective regions cannot always be guaranteed. Regarding watertightness, our method exhibits a degree of robustness when computing outward offsets on non-watertight meshes. This is because, even in the presence of open boundaries, the winding number algorithm has a low probability of misclassifying exterior triangles as interior. In contrast, inward offsets are more susceptible: interior triangles near boundaries are frequently misclassified as exterior, leading to fragmentation (see Figure 16) or even large-scale omission of regions incorrectly deemed outside the volume (Figure 17). In the presence of self-intersections, an outward offset can produce a repaired surface that eliminates the intersections and removes internal redundant faces (Figure 18). Conversely, an inward offset tends to produce broken or discontinuous geometry (Figure 19), as illustrated schematically in Figure 20. For non-manifold configurations, the quadratic optimization used in our method fails to recover sharp features, as analyzed in Figure 21. This leads to wrinkled or creased surfaces. Outward offsets on such inputs yield an enclosing but wrinkled outer surface, while inward offsets result in both wrinkling and fractures—due to limitations of the winding number computation near non-manifold structures.

4.4 Comparisons

Comparison Dataset. To facilitate comparative analyses, we constructed a comparison dataset. The compilation process entailed initially performing random sampling of a preliminary model set from Thingi10K [Zhou and Jacobson 2016b], followed by iterative refinement through supplementary selection protocols. These protocols were designed to ensure the resultant dataset simultaneously encompasses a representative diversity of model variations while maintaining compatibility with comparative algorithms capable of producing viable outputs within an acceptable success threshold. The comparison dataset contains models with varying topology and geometry complexities, In the dataset, there are 50 models with fewer than 10,00 triangles. There are 32 models with a triangle count ranging from 1,000 to 5,000. Additionally, there are 13 models with a triangle count between 5,000 and 8,000. Lastly, there are 5 models with more than 8,000 triangles. 7 models are non-manifold. 6 models are with open surface. The average number of genera, disconnected components, intersecting triangle pairs and holes are 2.4, 1.94, 13.4, and 0.227, respectively.

For arbitrary input meshes, our generated offsets are guaranteed to be self-intersection-free and degenerate-free under exact representations. If the input mesh M_I is watertight and self-intersection-free, our generated M_O is guaranteed to be watertight.

We evaluate the geometry accuracy of our generated offset meshes by measuring how much of the distance between the offset mesh and input is different from the offset distance specified by users.

To demonstrate the effectiveness of our approach, we compare against five state-of-the-art competing approaches, i.e., [Alpha Wrap 2022], [Chen et al. 2023], [Minkowski sum 2023], [Zint et al. 2023], and [Jung et al. 2004], by batch processing the entire test dataset with varying offset distance settings, i.e., $d = 0.05\%l$, $d = 0.1\%l$, $d = 0.5\%l$, $d = 1\%l$ and $d = 5\%l$ both inwardly (s = -1) and outwardly (s = 1). We compute the single way metric $D_{point}(M_O, M_I)$, $D_{plane}(M_O, M_I)$ and $D_{angle}(M_O, M_I)$. Note that the corresponding metric from M_I to M_O may become invalid. For example, during inward offsetting, as the offset distance increases, the inward offset surface can diminish in certain regions—eventually vanishing entirely in extreme cases—as shown in Figure 26. We set a maximum execution time of one hour for each algorithm on each model. For processing time more than an hour, we automatically terminate the process and mark it as a failure. While we elaborate

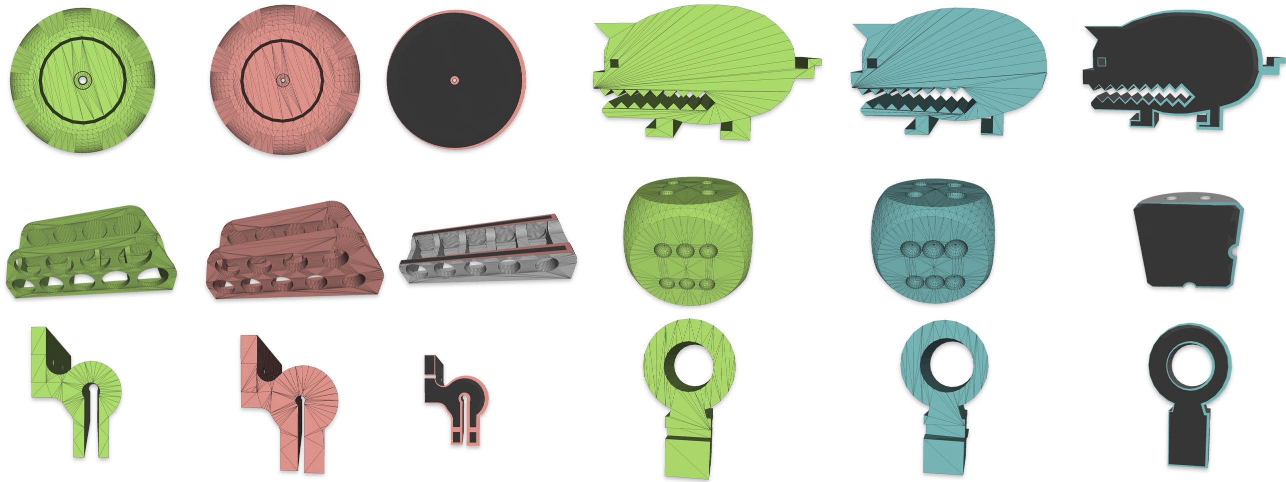


Fig. 14. Our approach supports the generation of feature-preserving non-uniform offsets, i.e., we interpolate between 0.01% and 2% of the bounding box diagonal length for each model, for any given mesh inputs (green), either inwardly (blue) or outwardly (red).

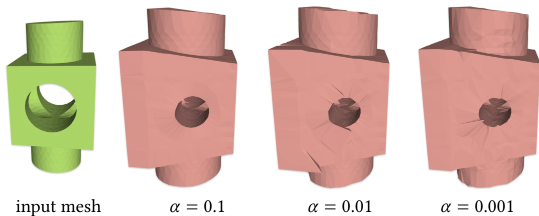


Fig. 15. Effect of α in non-uniform offset.

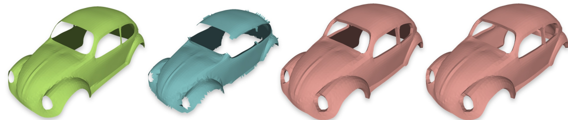


Fig. 16. This figure illustrates the offset operations applied to an open mesh. From left to right, the subfigures show the input model, the result of an inward offset, the result of an outward offset, and finally a shell structure formed by combining the outward-offset mesh with the input mesh. It can be observed that the inward offset operation produces a fragmented boundary due to constraints related to the winding number.

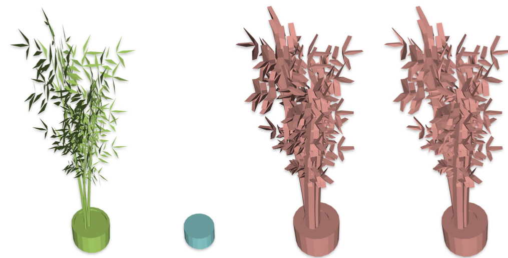


Fig. 17. This figure illustrates a sequence of offset operations applied to an open mesh. From left to right, the subfigures depict the input model, the inward offset result, the outward offset result, and a shell structure composed of the combined outward-offset and input meshes. It can be observed that the inward offset operation fails to preserve the leaf geometry of the input model due to constraints imposed by the winding number criterion.

the detailed comparisons below, Figures 11 and 23 show visual comparisons for outward offset results generated by the various methods, Figures 12 and 24 illustrate the visual differences for inward offset meshes produced by the comparing approaches, and Table 1 demonstrates all the quantitative statistics.

[Alpha Wrap 2022] can robustly generate a watertight and orientable surface triangle mesh from an arbitrary 3D geometry input, but cannot compute inward offsets. Therefore, as shown in Table 1, we compare with it for cases only when $s = 1$. It has two parameters, i.e., an alpha to determine the size of untraversable cavities when refining and carving a 3D Delaunay triangulation and an offset distance to control how far of the output mesh vertices from the input. We compare our algorithm with [Alpha Wrap 2022]'s CGAL implementation, set its offset distance using d. Its alpha parameter plays a critical role in determining the output mesh's geometry approximation of the input and the approach's computational speed. The mathematical definition of α lacks a direct correlation with the diagonal length of the mesh. Following the experimental setup of [Alpha Wrap 2022], we fix $\alpha = 100$ for all tests in this article. As α increases, the number of triangles grows significantly. While all evaluation metrics either improve or exhibit negligible change, this enhancement in metrics is achieved at the cost of a substantial increase in mesh face count, as demonstrated in Figure 25. Our analysis reveals that when the offset distance is large, this method performs better on the D_{point} metric. In contrast, our approach consistently achieves the smallest error for $D_{plane}(M_O, M_I)$ across all evaluated offset distances. In fact, our method excels at capturing the sharp features of the input geometry, demonstrating the best feature preservation performance among all compared

Table 1. This Table Presents the Distance Metrics (Point-to-Plane and Point-to-Point) and Angle Metrics for All Evaluated Methods

d	s	method	$\frac{D_{plane}(M_O,M_I)}{d}$	$\frac{D_{point}(M_O,M_I)}{d}$	$D_{angle}(M_O, M_I)$	SUC	FACE	TIME
0.05%l	-1	[Jung et al. 2004]	0.1116	0.1069	1.5453	95	**1938.6**	281.5
		[Chen et al. 2023]	-	-	-	0	-	-
		Ours	**0.0045**	**0.0054**	**0.1812**	**100**	2331.2	**8.3**
0.1%l	-1	[Jung et al. 2004]	0.1426	0.1371	2.5342	95	**1955.3**	203.9
		[Chen et al. 2023]	-	-	-	0	-	-
		Ours	**0.0049**	**0.0055**	**0.2533**	**100**	2348.2	**7.2**
0.5%l	-1	[Jung et al. 2004]	0.0906	0.0872	7.9583	94	**2010.2**	235.4
		[Chen et al. 2023]	0.0099	**0.0024**	2.8931	93	643734	70.4
		Ours	**0.0084**	0.0095	**0.8762**	**100**	2572.4	**14.6**
1%l	-1	[Jung et al. 2004]	0.1087	0.1036	12.3374	88	**1869.9**	236.3
		[Chen et al. 2023]	0.0221	**0.0040**	5.8442	**100**	172471	46.8
		Ours	**0.0129**	0.0147	**2.0013**	**100**	2922	**32.7**
5%l	-1	[Jung et al. 2004]	0.1644	0.1596	28.4349	58	1238	539.9
		[Chen et al. 2023]	0.1104	0.0244	13.4986	95	8459	**1.5**
		Ours	**0.0236**	**0.0184**	**7.5102**	**100**	**889**	117.4
0.05%l	1	[Alpha Wrap 2022]	0.1733	0.1890	9.7074	**100**	21239	**6**
		[Jung et al. 2004]	0.1373	0.1175	1.3588	95	**1898.6**	320.5
		[Chen et al. 2023]	-	-	-	0	-	-
		[Zint et al. 2023]	0.0614	0.0610	6.1789	14	9334070	104.277
		[Minkowski sum 2023]	1.1718	1.3165	5.8753	79	4967.2	425.1
		Ours	**0.0074**	**0.0188**	**0.3102**	**100**	2312.1	**6.1**
0.1%l	1	[Alpha Wrap 2022]	0.0675	0.0773	12.4270	**100**	17915.2	**5.1**
		[Jung et al. 2004]	0.2337	0.1911	2.1913	93	**1860.5**	209.2
		[Chen et al. 2023]	-	-	-	0	-	-
		[Zint et al. 2023]	0.0338	0.0305	2.5426	38	706214	247.291
		[Minkowski sum 2023]	0.4086	0.4584	6.3014	79	4965.8	404.5
		Ours	**0.0090**	**0.0192**	**0.4112**	**100**	2333.3	**6.4**
0.5%l	1	[Alpha Wrap 2022]	0.0395	0.0222	20.6634	**100**	15416	**4**
		[Jung et al. 2004]	0.2365	0.1902	6.4390	88	**1734.9**	294.8
		[Chen et al. 2023]	0.0330	0.0076	6.7319	90	570871	82.6
		[Zint et al. 2023]	0.0272	**0.0062**	2.1996	77	180346	176.7
		[Minkowski sum 2023]	0.1294	0.0390	6.8082	79	4858.7	422.9
		Ours	**0.0140**	0.0124	**1.0145**	**100**	2554.1	14.2
1%l	1	[Alpha Wrap 2022]	0.0589	0.0163	22.9292	**100**	15900.4	**3.4**
		[Jung et al. 2004]	0.2282	0.1778	9.5413	86	**1761**	438.2
		[Chen et al. 2023]	0.0615	0.0101	8.5363	**100**	126664	50.3
		[Zint et al. 2023]	0.0533	**0.0072**	4.7115	94	112318	160.5
		[Minkowski sum 2023]	0.1451	0.0314	7.8576	79	4639.1	403.3
		Ours	**0.0437**	0.0453	**1.2406**	**100**	3021	36.9
5%l	1	[Alpha Wrap 2022]	0.1433	**0.0103**	19.9315	**100**	20533.5	**2.7**
		[Jung et al. 2004]	0.1851	0.1256	18.7543	60	**1319.8**	780
		[Chen et al. 2023]	0.1501	0.0192	14.0047	**100**	4862.7	**1.5**
		[Zint et al. 2023]	0.1471	0.0141	13.8445	97	11977.9	45.6
		[Minkowski sum 2023]	0.1744	0.0351	14.7866	79	3301.1	418.7
		Ours	**0.1010**	0.0358	**4.9913**	94	3066.3	246.1

Our method demonstrates a stronger tendency to satisfy the point-to-plane distance metric compared to the point-to-point distance. Notably, methods producing a higher number of faces exhibit an advantage in meeting the point-to-point distance criterion. The angle column reflects the dihedral angle variation of sampling points along feature lines, highlighting our method's superior performance in feature preservation. The last three columns summarize: **SUC** (number of models successfully completed), **FACE** (total faces generated), and **TIME** (average time consumption). For reference, the six timeout models of our method required the following times: 3603s, 40005s, 5540s, 4477s, 13355s, and 3679s.

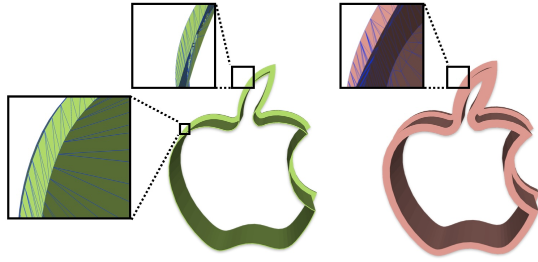


Fig. 18. This figure demonstrates the remediation of a self-intersecting model via an outward offset operation. The model is composed of a base plate and a curved, annular cylindrical structure. Although these two components intersect spatially, they are topologically disjoint. The left subfigure shows that the intersecting surfaces of the base and the curved cylinder are in close contact, resulting in rendering artifacts known as Z-fighting. The right subfigure demonstrates that after applying an outward offset, the internal intersecting surfaces are effectively removed, resulting in a clean, watertight geometry.

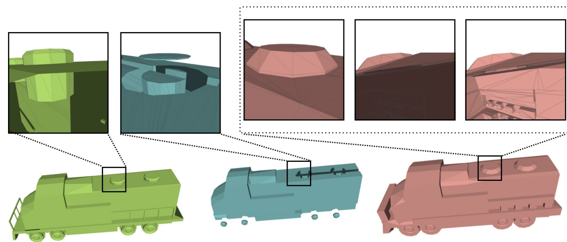


Fig. 19. This figure presents a case study of offset operations applied to a self-intersecting mesh. The primary sequence (from left to right) displays the input model, the inward offset, and the outward offset. The inward offset yields a fracture result due to the self-intersections in the input geometry. Furthermore, a detailed magnified view of a specific region on the outward offset result is provided. This close-up is decomposed into three components: the surface geometry, a cutaway view revealing the internal structure, and a highlighted representation of the internal cutaway results.

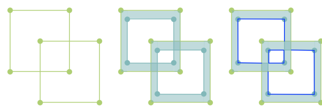


Fig. 20. This 2D schematic demonstrates the fragmentation of an inward offset on a self-intersecting mesh. From left to right: the input mesh; the planar region (light blue) created by the offset; and the resulting offset curve (dark blue), which fractures into three separate pieces.

approaches across all tested offset distances, as evidenced by the leading $D_{angle}(M_{\text{O}}, M_{\text{I}})$ scores in Table 1 and the visualizations in Figure 11.

Comparison with [Chen et al. 2023]. We compare our algorithm with [Chen et al. 2023] by directly running their provided online executable program. While [Chen et al. 2023] can robustly handle general inputs, relying on voxel discretization of the unsigned distance function and maching-cube-like linear approximation of the iso-surface, [Chen et al. 2023] can generate results only when

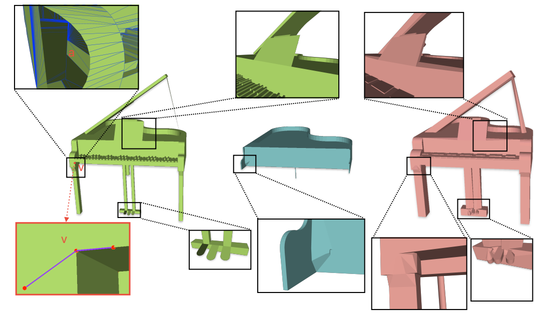


Fig. 21. This figure illustrates a case of a non-manifold mesh, showing from left to right the input model, the inward offset result, and the outward offset result. The red box shows the vertex generated by the vertex v being offset outward. Specifically, the connection between the music stand and the piano body creates a non-manifold edge. The left support pillar of the piano exhibits a highly complex internal triangle configuration with non-manifold topological connections. It can be observed that the vertex v in the diagram is adjacent to a triangle a, which is essentially an internal face connected via a non-manifold edge. This face supplies vertex v with a normal vector that is entirely opposite to the local surface of the mesh at that region. Consequently, our equation fails to compute a reasonable offset position correctly (highlighted in the red box within the figure), ultimately leading to fold-over artifacts in the final result. Furthermore, at the pedal assembly, one pedal is connected via a non-manifold edge and displays an incorrect normal vector. It should be noted that the model contains numerous other irregularities, only a subset of which are highlighted here.

Fig. 22. A visual gallery of the comparison dataset.

the discretization is coarse as shown by the $D_{point}(M_{\text{O}}, M_{\text{I}})$ for $d = 0.5\%l, s = -1$, $d = 1\%l, s = -1$ and $d = 0.5\%l, s = 1$. However, the provided implementation would incur memory and computational issues for small offset distances, which is denoted as "–" in Table 1 for program crashes. Although introducing the feature preserving EMC33 scheme, their results tend to produce creases, leading to noisy feature curves as illustrated in Figures 11 and 12.

Comparison with [Zint et al. 2023]. We compare our algorithm with [Zint et al. 2023] by running directly the source code provided by the authors. Their method also generates the mesh through implicit iso-surface extraction, similar to [Chen et al. 2023]. By using

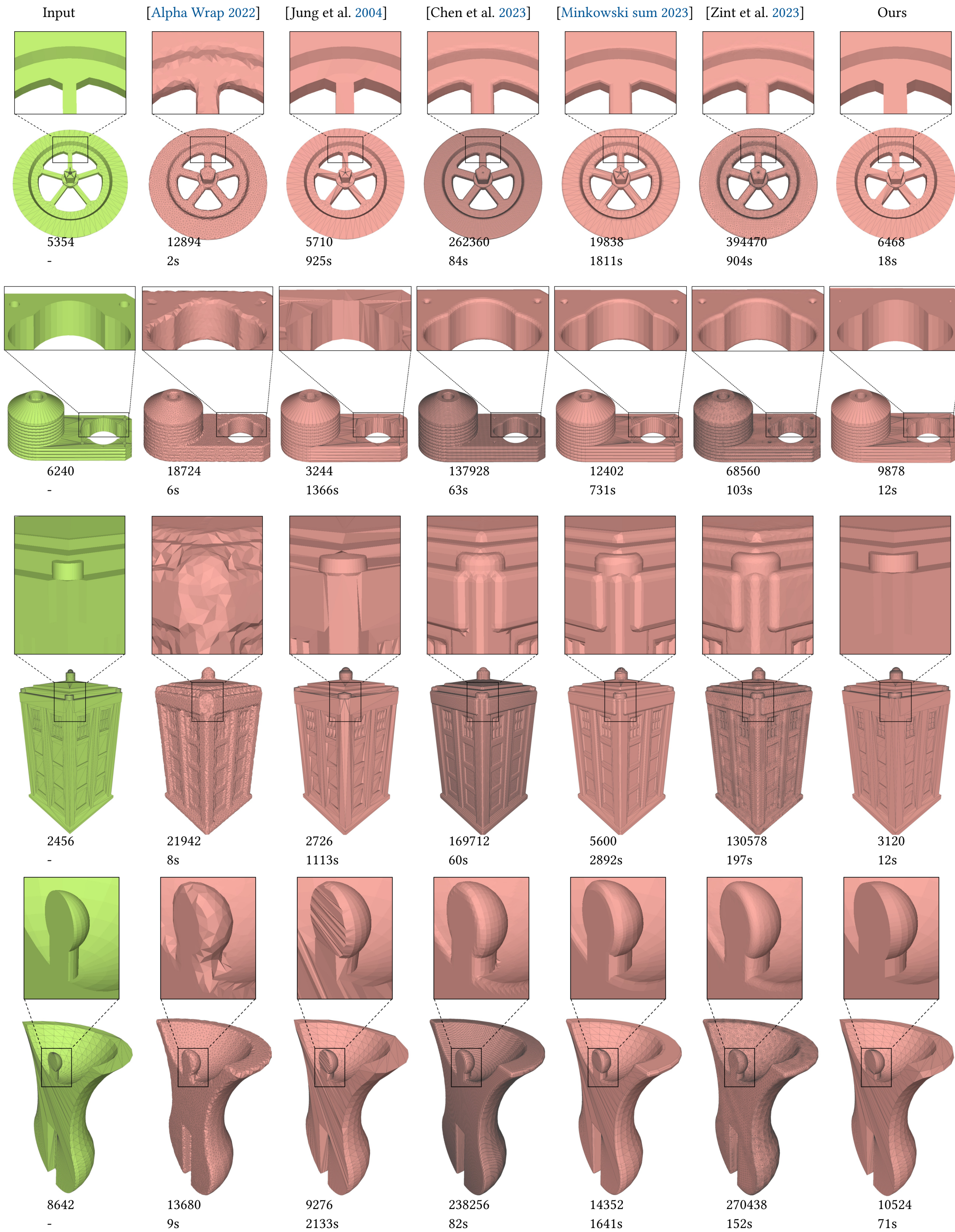


Fig. 23. This figure shows a comparison of our method with other methods for outward offsets. It is clearly visible that our method has an advantage in restoring sharp features and achieves better detail restoration. The two value recorded in each sub-figure are generating facets number and running time.

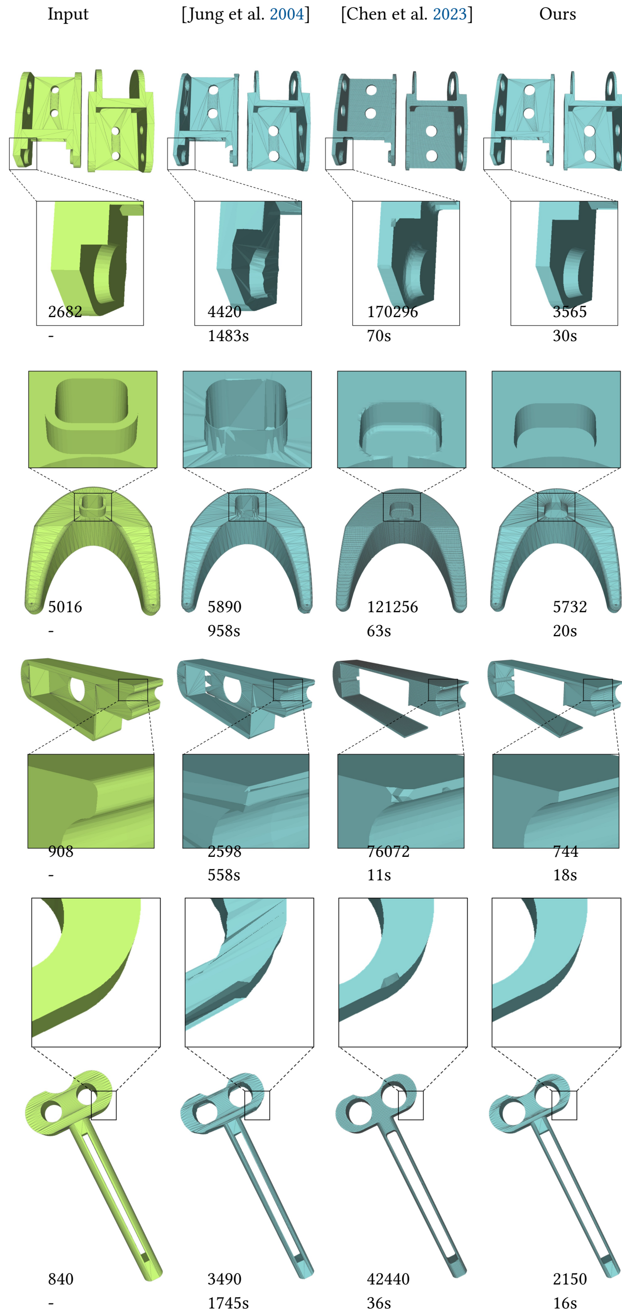
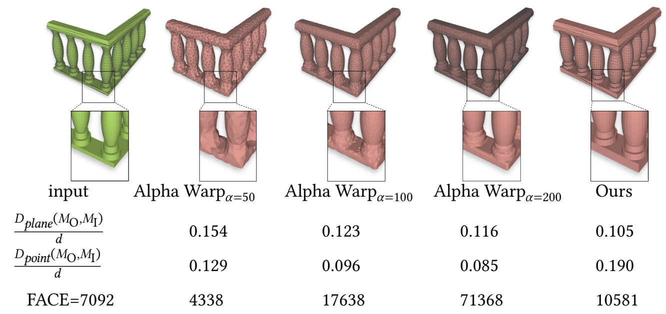


Fig. 24. This figure shows a comparison of our method with other methods for inward offsets. It is clearly visible that our method has an advantage in restoring sharp features and achieves better detail restoration. And [Jung et al. 2004] can cause the error facet like show in the third row. The two value recorded in each sub-figure are generating facets number and running time.



	input	Alpha Warp$_{\alpha=50}$	Alpha Warp$_{\alpha=100}$	Alpha Warp$_{\alpha=200}$	Ours
$\frac{D_{plane}(M_O, M_I)}{d}$		0.154	0.123	0.116	0.105
$\frac{D_{point}(M_O, M_I)}{d}$		0.129	0.096	0.085	0.190
	FACE=7092	4338	17638	71368	10581

Fig. 25. This figure illustrates the evaluation result of [Alpha Wrap 2022] and ours across varying alpha values.

octree, it can improve performance at small offset distances. The results of this algorithm are controlled by three main parameters: the minimum octree depth (d0), the maximum octree depth (d1), and the octree depth for resolving non-manifold vertices (d2). We use the default setting, i.e., d0 = 0, d1 = 10, and d2 = 12. Due to the usage of dual contouring for iso-surface extraction, this approach preserves feature lines very well, only second to our algorithm as show in the $D_{angle}(M_O, M_I)$ column of Table 1 and as show in Figure 11. It can get the best or close to the best result in $D_{point}(M_O, M_I)$ metric. As the offset distance gets small, the face numbers of M_O can reach to millions of triangles, e.g., over 9M for $d = 0.05\%l$. Furthermore, their provided program never successfully processes the entire dataset for the different batch tests. Due to unknown issues, the code provided by [Zint et al. 2023] cannot handle inward offsets, therefore, we only perform the outward offset experiments.

Comparison with [Minkowski sum 2023]. This method employs the computational results of the Minkowski Sum of a sphere (represented as a polyhedron) and M_I. It has a corresponding function available in version 5.6 of the CGAL library, released in 2023, and exclusively supports outward offsets. Instead of related to offset distances, the computational performance of this method is unstable and heavily depends on the model's concavity and convexity, with longer computation times observed for models with more concave features. If a model strictly adheres to the definition of a convex hull, the computational efficiency is high. It does not support meshes with self-intersections, open boundaries, and so on. The evaluated measurements for D_{point} and D_{plane} for this approach do not exhibit particularly good results. Its output is represented by planar polygons, which, when converted into a triangular mesh, shows a lower number of faces compared to those obtained by the implicit iso-surface extraction methods. This approach preserves feature lines well, as shown in Figure 11, but does not maintain dihedral angles well, as indicated in the $D_{angle}(M_O, M_I)$ column of Table 1.

Comparison with [Jung et al. 2004]. Using C++ and CGAL, we implemented the algorithm presented in [Jung et al. 2004]. This method achieves the offset surface by first directly moving vertices of M_I following their normal directions, and then calculating a convex hull of all of the moving points. However, their approach can lead to missing or erroneous faces as shown in the second column of Figure 24. Moreover, in scenarios with complex and many self-intersections, this approach can have a significant increase in computational time as the offset distance increases, whereas our method does not show such a significant rise as shown in the Table 1 and Figure 32 when offset distance become large like $d >= 1\%l$. For many cases, relying solely on the normal direction can easily

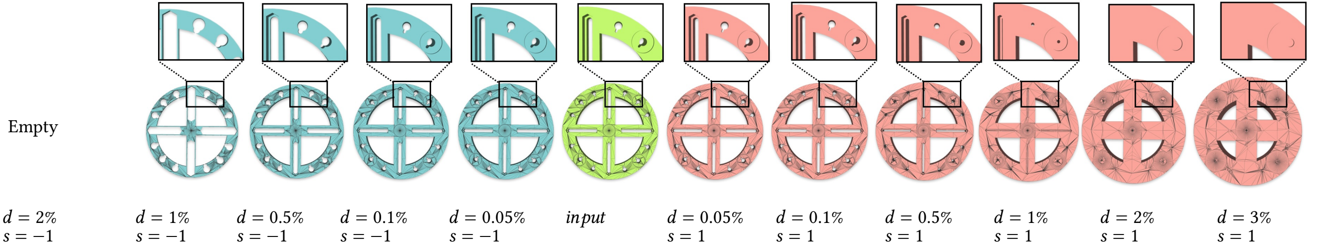


Fig. 26. Show the mesh in different offset to generate a empty mesh.

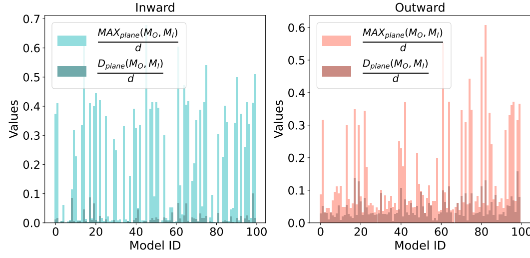


Fig. 27. A comparison between $\frac{MAX_{plane}(M_O,M_I)}{d}$ and $\frac{D_{plane}(M_O,M_I)}{d}$ is performed using a 1% inward and outward offset.

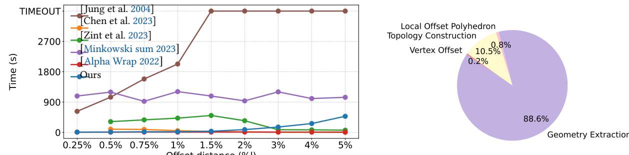


Fig. 28. The left part this figure shows the different method timing cost with 5 representative models. In 0.25%l offset distance, [Zint et al. 2023] and [Chen et al. 2023] can not success generate the offset mesh. The right part shows the execution time of each part of our algorithm.

produce a large number of creases, thus generating many redundant features as shown in Figures 11 and 12. Additionally, this results in particularly poor D_{angle} values. And its D_{point} and D_{plane} values often among the worst over all algorithm as shown in Table 1.

To summarize, our method can excellently capture sharp features of the inputs and has the best $D_{plane}(M_O, M_I)$ and $D_{angle}(M_O, M_I)$ among all methods because our approach focuses more on the mitered offset effect. Compared to the competing approaches, our method runs fast at small offset distances, and due to the presence of acceleration strategies, it still performs reasonably well as the offset distance increases. For a more comprehensive evaluation, a metric $\frac{MAX_{plane}(M_O,M_I)}{d}$ is introduced.

$$
\begin{aligned}
H_{\max}(M_O, M_I) &= \max_{v \in M_O} \{d_t(v, M_I)\}, \\
MAX_{plane}(M_O, M_I) &= |H_{max}(M_O, M_I) - d|,
\end{aligned} \tag{8}
$$

A comparison between $\frac{MAX_{plane}(M_O,M_I)}{d}$ and $\frac{D_{plane}(M_O,M_I)}{d}$ is performed using a $d = 1\%$ inward and outward offset as an example, as illustrated in Figure 27. It can be observed $\frac{MAX_{plane}(M_O,M_I)}{d}$ metric also yields a very small value. While the $\frac{MAX_{plane}(M_O,M_I)}{d}$ metric coincides with $\frac{D_{plane}(M_O,M_I)}{d}$ in simple cases, it remains a small value under complex scenarios.

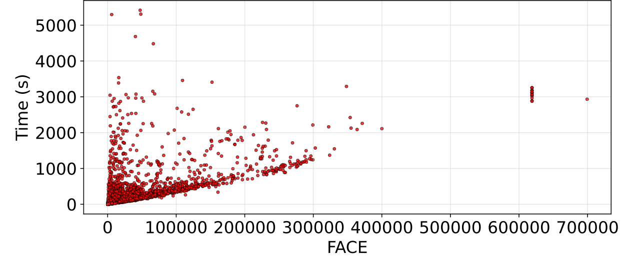


Fig. 29. Time(s) vs. FACE of all model completed in time limit of Thingi10K.

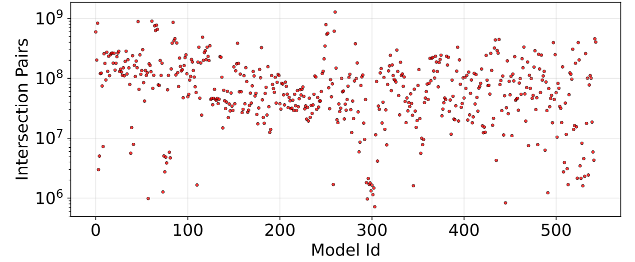


Fig. 30. Number of intersection pairs in failed models.

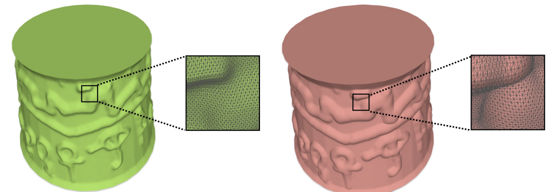


Fig. 31. This figure depicts a model that failed to complete within the time limit. Given its high density (619,008 triangles), the computational complexity for intersection and ray-casting checks became prohibitively high, making it hard to extract the offset surface.

4.5 Timing

As shown in the T column of Table 1, averaged over the entire tested dataset, our approach achieves the fastest computational speed for inward offset generation when offset distance $\leq 1\%$. For

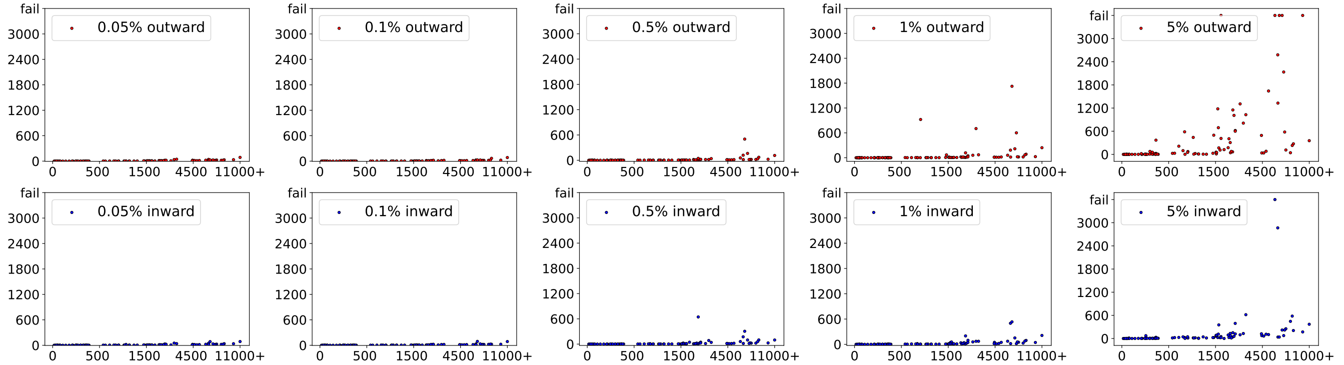


Fig. 32. Computation time (in seconds) vs. Triangle count for our approach. Data Points exceeding 3,600 seconds are marked as fail.

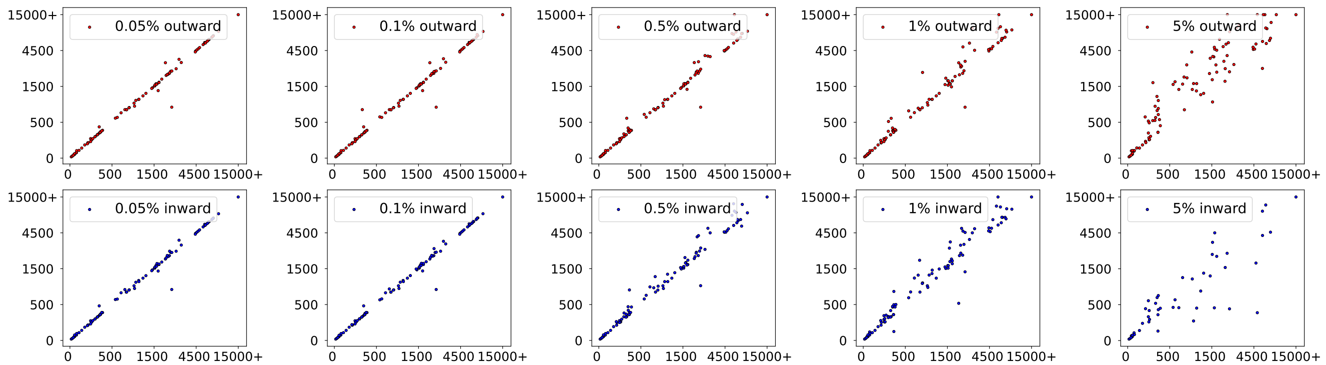


Fig. 33. Triangle number of output vs. input generated by our approach.

outward offset generation, while [Portaneri et al. 2022] is consistently among the fastest approach, as the offset distance increases, ours changes from being comparable to [Portaneri et al. 2022] at 0.05% and 0.1% and is more than 30 times faster compared to [Jung et al. 2004], [Zint et al. 2023], and [Minkowski sum 2023], more than 10 times faster compared to [Jung et al. 2004], [Chen et al. 2023], [Zint et al. 2023], and [Minkowski sum 2023] when $d = 0.5\%$, about 10 times faster compared to [Jung et al. 2004], and [Minkowski sum 2023] when $d = 1\%$, and only 2-3 times faster than [Jung et al. 2004], and [Minkowski sum 2023] when $d = 5\%$. Top-left corner of Figure 32 further compares the competing approaches with ours on five selected representative models by varying the offset distances from $d = 0.25\%$ to 5%. It's clear that our approach ranks among the fastest when offset distance is small and the computation slows down for large offset distances. The reason behind is due to the increased amount of self-intersection computations of our approach when d increases. Bottom-left corner of Figure 32 further verifies that our computational bottleneck lies in the intersection resolving operations for models with excessive details. This step accounts 88.6% of the total computing time. Figure 32 plots the detailed timing for each model by our approach for the varying offset distances.

To further evaluate the performance of our method, we performed comprehensive evaluations on the entire Thingi10K dataset. Due to computational resource constraints, a maximum runtime limit of 3 hours per model was imposed, and only a d = 0.01% outward offset was applied. Among the 9,994 models tested, 544 exceeded the time limit (Figure 29). For these failed cases, we quantified the number of intersecting face pairs, as summarized in Figure 30. A noteworthy finding is the catastrophic increase in computational complexity when extensive intersections occur. This leads to a distinct runtime dichotomy: models either succeed relatively quickly (under 1.5 hours) or fail by hitting the time limit, with a near-total absence of failures in the 1.5–3 hour range.

Theoretically, our method is capable of performing offset calculations for arbitrary meshes. however, in practical testing scenarios, it fails to yield results within the constrained time and computational resources for highly complex models (e.g., those with intricate self-intersections or a large number of mesh faces). As illustrated in Figure 31, the model successfully computes the Offset Polyhedra but fails to extract the offset surface within the allotted time limit.

5 Conclusion

We propose a new and robust approach to generate an offset mesh for a 3D input with an arbitrary topology and geometry complexities. Our method ensures the output with several nice properties, such as feature preservation, similar number of triangles with the input, free of self-intersections and degenerate elements. Our approach also support user-specified non-uniform offset distances.

We anticipate that, with all these advantages combined, our method makes a significant advancement in the geometry processing related field.

Discussion and Limitations: Several aspects worth to investigate to further improve the current approach. First, the generalized winding number may be computed wrongly for points very close to the mesh surface, which is an issue could be mitigated through the combination of ray-intersection checks. Second, to introduce as few as possible approximations to the offset mesh generation, we only consider conservative speedup strategies in the current version, where for applications in rendering and animation, we may further improve the performance by designing accelerations with a controllable tolerance to the geometric distance.

References

2024. The OEIS Foundation Inc. Retrieved May 1, 2024 from https://oeis.org/search?q=A101052&language=english&go=Search

James W. Alexander. 1922. A proof and extension of the Jordan-Brouwer separation theorem. *Transactions of the American Mathematical Society* 23, 4 (1922), 333–349.

Nina Amenta, Sunghee Choi, and Ravi Krishna Kolluri. 2001. The power crust, unions of balls, and the medial axis transform. *Computational Geometry* 19, 2-3 (2001), 127–153.

Alan Brunton and Lubna Abu Rmaileh. 2021. Displaced signed distance fields for additive manufacturing. *ACM Transactions on Graphics* 40, 4 (2021), 1–13.

Marcel Campen and Leif Kobbelt. 2010. Polygonal boundary evaluation of minkowski sums and swept volumes. *Computer Graphics Forum* 29, 5 (2010), 1613–1622.

CGAL(v5.6). [n. d.]. *CGAL User and Reference Manual* (5.6 ed.). CGAL Editorial Board. Retrieved from https://doc.cgal.org/5.6/Manual/packages.html

Lufeng Chen, Man-Fai Chung, Yaobin Tian, Ajay Joneja, and Kai Tang. 2019a. Variable-depth curved layer fused deposition modeling of thin-shells. *Robotics and Computer-Integrated Manufacturing* 57 (2019), 422–434.

Zhen Chen, Zherong Pan, Kui Wu, Etienne Vouga, and Xifeng Gao. 2023. Robust low-poly meshing for general 3D models. *ACM Transactions on Graphics* 42, 4 (2023), 1–20. DOI: https://doi.org/10.1145/3592396

Zhen Chen, Daniele Panozzo, and Jeremie Dumas. 2019b. Half-space power diagrams and discrete surface offsets. *IEEE Transactions on Visualization and Computer Graphics* 26, 10 (2019), 2970–2981.

P. Cignoni, C. Rocchini, and R. Scopigno. 1998. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum* 17, 2 (1998), 167–174.

Xifeng Gao, Hanxiao Shen, and Daniele Panozzo. 2019. Feature preserving octree-based hexahedral meshing. *Computer Graphics Forum* 38, 5 (2019), 135–149. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13795 DOI: https://doi.org/10.1111/cgf.13795

Samuele Giraudo. 2015. Combinatorial operads from monoids. *Journal of Algebraic Combinatorics* 41 (2015), 493–538.

André Guéziec and Robert Hummel. 2002. Exploiting triangulated surface extraction using tetrahedral decomposition. *IEEE Transactions on Visualization and Computer Graphics* 1, 4 (2002), 328–342.

Peter Hachenberger. 2009. Exact Minkowksi sums of polyhedra and exact and efficient decomposition of polyhedra into convex pieces. *Algorithmica* 55, 2 (2009), 329–345.

Peter Hachenberger, Lutz Kettner, and Kurt Mehlhorn. 2007. Boolean operations on 3D selective Nef complexes: Data structure, algorithms, optimized implementation and experiments. *Computational Geometry* 38, 1 (2007), 64–99. DOI: https://doi.org/10.1016/j.comgeo.2006.11.009. Special Issue on CGAL.

Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo. 2018. Tetrahedral meshing in the wild. *ACM Transactions on Graphics* 37, 4 (2018), 60–1.

Alec Jacobson, Ladislav Kavan, and Olga Sorkine-Hornung. 2013. Robust inside-outside segmentation using generalized winding numbers. *ACM Transactions on Graphics* 32, 4 (2013), 1–12.

Zhongshi Jiang, Teseo Schneider, Denis Zorin, and Daniele Panozzo. 2020. Bijective projection in a shell. *ACM Transactions on Graphics* 39, 6 (2020), 1–18.

Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. 2002. Dual contouring of hermite data. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. 339–346.

Wonhyung Jung, Hayong Shin, and Byoung K. Choi. 2004. Self-intersection removal in triangular mesh offsetting. *Computer-Aided Design and Applications* 1, 1-4 (2004), 477–484.

Leif P. Kobbelt, Mario Botsch, Ulrich Schwanecke, and Hans-Peter Seidel. 2001. Feature sensitive surface extraction from volume data. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. 57–66.

Louisa Lam, Seong-Whan Lee, and Ching Y. Suen. 1992. Thinning methodologies-a comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14, 09 (1992), 869–885.

Pan Li, Bin Wang, Feng Sun, Xiaohu Guo, Caiming Zhang, and Wenping Wang. 2015. Q-mat: Computing medial axis transform by quadratic error minimization. *ACM Transactions on Graphics* 35, 1 (2015), 1–16.

Shengjun Liu and Charlie C. L. Wang. 2010. Fast intersection-free offset surface generation from freeform models with triangular meshes. *IEEE Transactions on Automation Science and Engineering* 8, 2 (2010), 347–360.

Takashi Maekawa. 1999a. An overview of offset curves and surfaces. *Computer-Aided Design* 31, 3 (1999), 165–173. DOI: https://doi.org/10.1016/S0010-4485(99)00013-5

Takashi Maekawa. 1999b. An overview of offset curves and surfaces. *Computer-Aided Design* 31, 3 (1999), 165–173.

Wenlong Meng, Shuangmin Chen, Zhenyu Shu, Shi-Qing Xin, Hongbo Fu, and Changhe Tu. 2018. Efficiently computing feature-aligned and high-quality polygonal offset surfaces. *Computers and Graphics* 70 (2018), 62–70.

Sang C. Park and Yun C. Chung. 2003. Mitered offset for profile machining. *Computer-Aided Design* 35, 5 (2003), 501–505.

Darko Pavić and Leif Kobbelt. 2008. High-resolution volumetric computation of offset surfaces with feature preservation. In *Proceedings of the Computer Graphics Forum*. Wiley Online Library, 165–174.

B. Pham. 1992. Offset curves and surfaces: A brief survey. *Computer-Aided Design* 24, 4 (1992), 223–229. DOI: https://doi.org/10.1016/0010-4485(92)90059-J

Cédric Portaneri, Mael Rouxel-Labbé, Michael Hemmer, David Cohen-Steiner, and Pierre Alliez. 2022. Alpha wrapping with an offset. *ACM Transactions on Graphics* 41, 4 (2022), 1–22.

QGIS Development Team. 2024. Vector geometry algorithms. Retrieved June 1, 2025 from https://docs.qgis.org/3.4/en/docs/user_manual/processing_algs/qgis/vectorgeometry.html. QGIS Documentation.

Jaroslaw R. Rossignac and Aristides A. G. Requicha. 1986. Offsetting operations in solid modelling. *Computer Aided Geometric Design* 3, 2 (1986), 129–148.

B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. 2020. OSQP: An operator splitting solver for quadratic programs. *Mathematical Programming Computation* 12, 4 (2020), 637–672. DOI: https://doi.org/10.1007/s12532-020-00179-2

Feng Sun, Yi-King Choi, Yizhou Yu, and Wenping Wang. 2013. Medial meshes for volume approximation. arXiv:1308.3917. Retrieved from https://arxiv.org/abs/1308.3917

Feng Sun, Yi-King Choi, Yizhou Yu, and Wenping Wang. 2015. Medial meshes–a compact and accurate representation of medial axis transform. *IEEE Transactions on Visualization and Computer Graphics* 22, 3 (2015), 1278–1290.

Andrea Tagliasacchi, Thomas Delame, Michela Spagnuolo, Nina Amenta, and Alexandru Telea. 2016. 3d skeletons: A state-of-the-art report. 35, 2 (2016), 573–597.

Wayne Tiller and Eric G. Hanson. 1984. Offsets of two-dimensional profiles. *IEEE Computer Graphics and Applications* 4, 9 (1984), 36–46. DOI: https://doi.org/10.1109/MCG.1984.275995

Tim Van Hook. 1986. Real-time shaded NC milling display. *ACM SIGGRAPH Computer Graphics* 20, 4 (1986), 15–20.

Bala R. Vatti. 1992. A Generic Solution to Polygon Clipping. *Communications of the ACM* 35, 7 (1992), 56–63.

Charlie C. L. Wang and Yong Chen. 2013. Thickening freeform surfaces for solid fabrication. *Rapid Prototyping Journal* (2013).

Charlie C. L. Wang and Dinesh Manocha. 2013. GPU-based offset surface computation using point samples. *Computer-Aided Design* 45, 2 (2013), 321–330.

Qingnan Zhou and Alec Jacobson. 2016a. Thingi10K: A dataset of 10,000 3D-printing models. arXiv:1605.04797. Retrieved from https://arxiv.org/abs/1605.04797

Qingnan Zhou and Alec Jacobson. 2016b. Thingi10k: A dataset of 10,000 3d-printing models. arXiv:1605.04797. Retrieved from https://arxiv.org/abs/1605.04797

Daniel Zint, Zhouyuan Chen, Yifei Zhu, Denis Zorin, Teseo Schneider, and Daniele Panozzo. 2025. Topological offsets. *ACM Transactions on Graphics* 44, 4 (2025), 1–19.

Daniel Zint, Nissim Maruani, M. Rouxel-Labb, and Pierre Alliez. 2023. Feature-Preserving Offset Mesh Generation from Topology-Adapted Octrees. In *Proceedings of the Computer Graphics Forum*. Wiley Online Library, e14906.

Received 7 August 2024; revised 7 February 2026; accepted 12 February 2026