Robust Low-Poly Meshing for General 3D Models

ZHEN CHEN, The University of Texas at Austin & LightSpeed Studios, USA
ZHERONG PAN, LightSpeed Studios, USA
KUI WU, LightSpeed Studios, USA
ETIENNE VOUGA, The University of Texas at Austin, USA
XIFENG GAO, LightSpeed Studios, USA

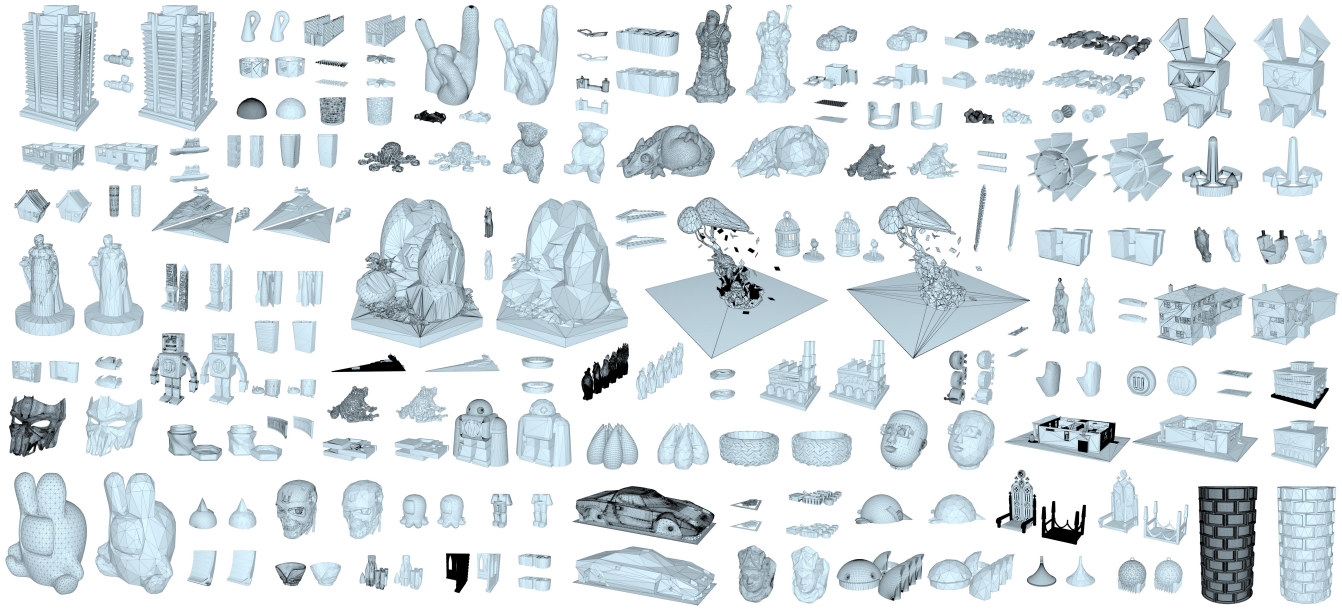


Fig. 1. A gallery of *wild* high-poly input meshes and their corresponding low-poly outputs generated by our method, where the low-polys are manifold, watertight, and self-intersection free, and have a small visual difference from their high-poly counterparts.

We propose a robust re-meshing approach that can automatically generate visual-preserving low-poly meshes for any high-poly models found in the wild. Our method can be seamlessly integrated into current mesh-based 3D asset production pipelines. Given an input high-poly, our method proceeds in two stages: 1) Robustly extracting an offset surface mesh that is feature-preserving, and guaranteed to be watertight, manifold, and self-intersection free; 2) Progressively simplifying and flowing the offset mesh to bring it close to the input. The simplicity and the visual-preservation of the generated low-poly is controlled by a user-required target screen size of the input: decreasing the screen size reduces the element count of the low-poly but enlarges its visual difference from the input. We have evaluated our method on a subset of the Thingi10K dataset that contains models created by practitioners in different domains, with varying topological and geometric complexities. Compared to state-of-the-art approaches and widely used software, our method demonstrates its superiority in terms of the element count, visual preservation, geometry, and topology guarantees of the generated low-polys.

CCS Concepts: • **Computing methodologies** → **Mesh geometry models**.

Additional Key Words and Phrases: iso-surface extraction, remeshing, geometry processing

ACM Reference Format:
Zhen Chen, Zherong Pan, Kui Wu, Etienne Vouga, and Xifeng Gao. 2023. Robust Low-Poly Meshing for General 3D Models. *ACM Trans. Graph.* 42, 4, Article 119 (August 2023), 20 pages. https://doi.org/10.1145/3592396

Authors' addresses: Zhen Chen, The University of Texas at Austin & LightSpeed Studios, Austin, Texas, USA, 78712, zchen96@cs.utexas.edu; Zherong Pan, LightSpeed Studios, Bellevue, Washington, USA, 98004, zrpan@global.tencent.com; Kui Wu, LightSpeed Studios, Los Angeles, California, USA, 90066, kwwu@global.tencent.com; Etienne Vouga, The University of Texas at Austin, Austin, Texas, USA, 78712, evouga@cs.utexas.edu; Xifeng Gao, LightSpeed Studios, Bellevue, Washington, USA, 98004, xifgao@global.tencent.com.

0730-0301/2023/8-ART119
https://doi.org/10.1145/3592396

1 INTRODUCTION

Mesh is a ubiquitously employed representation of 3D models for digital games. While a mesh with a large number of polygons (high-poly) is required to express visually appealing details, rendering its low-poly approximation at distant views is a typical solution to achieve real-time gaming experience, especially on low-end devices. High-polys, no matter whether they are manually created

through modeling software or automatically converted from CSG and implicit functions, often have complex topology and geometries, such as numerous components, high genus, non-manifoldness, self-intersections, degenerate elements, gaps, inconsistent orientations, etc. These complexities can pose significant challenges to the design of automatic low-poly mesh generation algorithms.

Over past decades, two typical ways are developed to obtain low-poly textured models: automatic *mesh reduction* that preserves original textures [Hoppe 1999; Liu et al. 2017]; manual mesh modeling followed by UV-generation and texture baking that creates new textures. Mesh reduction usually removes triangles through iterative application of local operations [Garland and Heckbert 1997] or element clustering [Cohen-Steiner et al. 2004], which relies on existing mesh vertices and connectivity. As a result, this method is only suitable for generating the medium-levels of LOD, while introducing serious artifacts when generating low-polys for meshes with excessive topology complexity as illustrated in Fig. 12 and Fig. 13. In the second pipeline, while UV-generation and texture baking can be done via semi-automatic tools (e.g. Maya and Marmoset), manual meshing is the most labor(cost)-intensive step. Therefore, we address this most urgent and challenging problem, aka, low-poly meshing.

Many automatic *re-meshing* approaches exist to represent the original mesh with a proxy and simplify the proxy mesh via a row of different techniques, e.g., polygonal mesh construction by plane fitting and mixed-integer optimization [Nan and Wonka 2017], cage mesh generation through voxel dilation and mesh simplification [Calderon and Boubekeur 2017], shape abstraction by feature simplification [Mehra et al. 2009], extremely low-poly meshing using visual-hull boolean operations [Gao et al. 2022], mesh simplification through differentiable rendering [Hasselgren et al. 2021], enclosing mesh generation through alpha-wrapping with an offset [Portaneri et al. 2022], and learning based approaches [Chen et al. 2020, 2022b]. However, these re-meshing approaches either rely on heavy user interactions, need careful parameter tweaking, or work for a limited type of model. Commercial software also has low-poly mesh functions, but generates unsatisfactory results in many cases. For example, in Fig. 11, and Table 3, we show the generated results using different low-poly construction modules in Simplyon [AB 2022]. It generates meshes with either triangle intersections, or non-satisfactory visual appearances. It remains to be a challenging task to automatically and robustly generate low-poly meshes for general 3D models used in the industry.

In practice, artists still manually craft low-poly meshes to ensure that they have a small number of triangles and preserve the visual appearance of the original mesh as much as possible. This often involves multiple iterations of manual adjustments, which incurs intensive labor work and prolonged project periods and remains to be a bottleneck for the current fast-changing game industry. Robust and automatic approaches that can generate satisfactory low-polys are in high demand.

From an input mesh with arbitrary topology and geometry properties, our goal is to generate its low-poly counterpart that is visually indistinguishable from faraway views. The visual appearance of a 3D shape can be evaluated by its silhouette and surface normal, while the simplicity of a low-poly mesh is typically measured by the number of triangles. We propose a new approach to generate low-polys that is both simple and visual-preserving, with the additional guarantees of being manifold, watertight, and self-intersection free. These additional properties are essential for artists to conveniently perform UV-generation and texture baking on the bare low-poly mesh.

Our method combines the idea of mesh-reduction and re-meshing. During the first stage, we re-mesh the high-poly to a "clean" proxy mesh and remove all the topological complexities therein. We then aggressively reduce the element count of the proxy mesh, while geometrically deforming the proxy to maintain visual preservation, leading to our low-poly output.

Our method specifically requires two inputs: a high-poly mesh and a parameter n_p, which represents the screen-space size of the high-poly mesh when rasterized onto a 2D screen. In practical terms, let l_p denote the 2D screen's pixel length, and l represent the diagonal length of the high-poly mesh's bounding box. The parameter n_p can be calculated as l/l_p, signifying the maximum number of pixels that the high-poly mesh's diagonal could occupy across all potential rendering views. During the first stage of our method, we build an unsigned distance field for the input and introduce a novel offset surface extraction method to extract a d-isosurface with $d = l/n_p$. Our algorithm not only guarantees the offset mesh is manifold, watertight, and self-intersection-free, but also recovers the normal approximated sharp features. During our second stage, we alternate among three steps, i.e. mesh simplification, mesh flow process, and feature alignment, to reduce the element count of the extracted mesh, while bringing the mesh close to the input and maintaining the aforementioned guarantees. All three steps contain only local operations, such as edge collapse and vertex optimization. Therefore, any local operation that violates a guarantee can be easily rolled back.

By construction, our algorithm is robust and automatic. The effectiveness of our approach is demonstrated by comparing it with state-of-the-art methods and popularly used software on a subset of the Thingi10K dataset [Zhou and Jacobson 2016] containing 3D models with varying complexities. All the data shown in the paper and the executable program can be found here[1].

2 RELATED WORKS

We first review low-poly mesh generation methods and then summarize the methods for iso-surface extraction.

2.1 Low-poly Meshing

Obtaining a low-poly mesh has been a research focus in computer graphics for several decades. Early works use various mesh reduction techniques that directly operate on the original inputs through iterative local element removal. Examples involve geometric error-guided techniques [Garland and Heckbert 1997; Hoppe 1996; Lescoat et al. 2020], structure-preserving-constrained technique [Salinas et al. 2015], volume-preserving technique [Lindstrom and Turk 1998], and image-driven technique [Lindstrom and Turk 2000], to name just a few. Clustering-based approaches [Cohen-Steiner et al. 2004; Li and Nan 2021] provide another direction for reducing the

[1] https://robust-low-poly-meshing.github.io/

element count. An inclusive survey is given in Khan et al. [2022]. While these approaches are well recognized in game production pipelines, they are better suited for reducing the mesh size of the original models to a medium level, e.g. reducing the number of faces by 20% - 80%. Unfortunately, for 3D graphics applications running on lower-end devices, often a much coarser low-poly mesh is desired. Such extremely low-poly meshes require topologic and geometric simplifications that are far beyond the capabilities of these mesh reduction techniques. Unlike mesh reduction techniques, a parallel effort aims at re-meshing, i.e., completely reconstructing a new mesh mimicking the original one. These methods vary drastically in their techniques and we classify them by their main feature into *voxelization base re-meshing*, *primitive fitting*, *visual-driven*, and *learning-based*.

Voxelization Based Re-meshing. Both Mehra et al. [2009] and Calderon and Boubekeur [2017] rely on a voxelization of the raw inputs to obtain a clean voxel surface. While Mehra et al. [2009] requires feature-guided re-triangulation, deformation, and curve-network cleaning to generate shape abstractions for architectural objects, Calderon and Boubekeur [2017] assumes the input meshes come with clear separation of the inside and outside space and heavily depends on user interactions to generate the final low-polys. Recently, Wu et al. [2022] combine voxelization-based remeshing with patch-based simplification to generate occluders for building models to pre-cull unseen meshes before online rendering.

Primitive Fitting. Various primitives can be composed to fit an object. For example, methods in [Bauchet and Lafarge 2020; Chauve et al. 2010; Fang and Lafarge 2020; Fang et al. 2018; Kelly et al. 2017; Nan and Wonka 2017] first compute a set of planes to approximate patch features detected in point clouds or 3D shapes, and then select a faithful subset of the intersecting planes to obtain the desired meshes. However, the key challenges of this type of method are: 1) properly computing a suitable set of candidate planes is already a hard problem by itself; 2) the complexity of the resulting mesh is highly unpredictable, requiring many trial-and-error to find a possible good set of parameters. Works using other primitives [Huang et al. 2014; Mehra et al. 2009; Wei et al. 2022; Yang and Chen 2021], such as boxes, convex shapes, curves, etc., are also promising directions, but none of them has been specifically dedicated for generating low-polys.

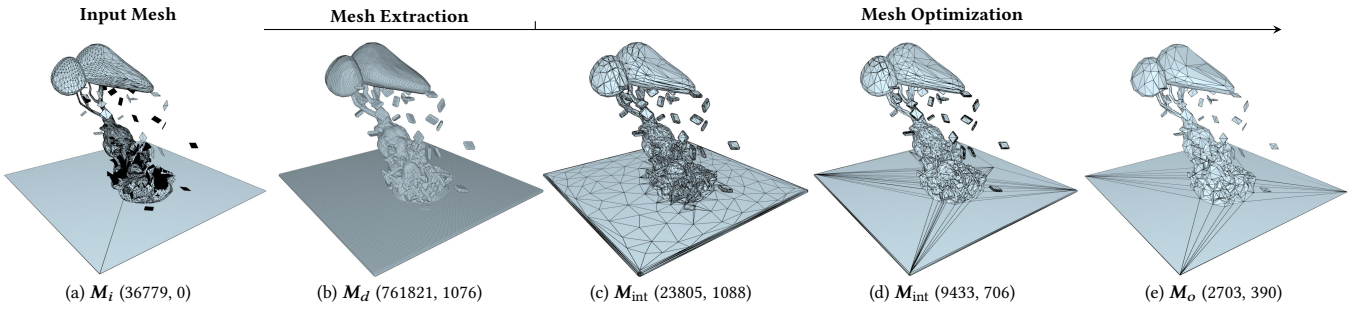
Visual-driven Approaches. Differentiable rendering [Laine et al. 2020] rises as a hot topic that enables the continuous optimization of scene elements through the guidance of rendered image losses. However, most of them [Hasselgren et al. 2021; Luan et al. 2021; Nicolet et al. 2021] require an initial mesh that is typically a uniformly discretized sphere. The key obstacle to generating low-polys via differentiable rendering is that the mesh reduction cannot be modeled as a differentiable optimization process. Although the analysis-by-synthesis type of optimizations [Luan et al. 2021] could be employed, the Laplacian regularization term used by most differentiable rendering techniques can guide the mesh far from the groundtruth, especially in a low-poly setting. A visual hull-based approach [Gao et al. 2022] has been recently proposed to generate extremely low-polys for building models, however, it not only creates sharp creases for organic shapes, but also makes it hard to control the target element number.

Learning-based Methods. A conventional 3D mesh reconstruction pipeline is composed of three steps: plane detection, intersection, and selection, while learning-based methods enable alternative pipelines. As an example, by converting it to a BSP-net, Chen et al. [2020] demonstrates that low-polys can be extracted from images. However, this method shares the common shortcomings of learning approaches: a large dataset is required for the network training, and the learned model works only for meshes of a similar type. It further requires the voxelizations of the dataset to have well-defined in/out segmentation. Furthermore, the generated meshes inherit the issues of polyfit-like [Nan and Wonka 2017] approaches: it creates sharp creases that are not present in the high-poly, and parameter tuning is difficult. By embedding a neural net of marching tetrahedral into the differentiable rendering framework, Munkberg et al. [2022] can optimize the meshes and materials simultaneously. As demonstrated in their work, by controlling the rendered image resolution, it can generate 3D models in a LOD manner. But these extended features brought by learning approaches are beyond the scope of our investigation.

2.2 Iso-surfacing Algorithms

The marching cubes (MC) algorithm was proposed concurrently by Lorensen and Cline [1987] and Wyvill et al. [1986] for reconstructing iso-surfaces from discrete signed distance fields. Several follow-up works were proposed to solve the tessellation ambiguities in each cube [Chernyaev 1995; Dürst 1988; Matveyev 1994; Nielson 2003, 2004; Nielson and Hamann 1991]. One of the best methods is MC33 [Chernyaev 1995], which enumerates all possible topologic cases based on trilinear interpolation in the cube. The follow-ups resolve non-manifold edges in MC33 [Custodio et al. 2013; Lopes and Brodlie 2003]. MC33 was correctly implemented by Vega et al. [2019] after resolving the defective issues of the previous implementations [Custodio et al. 2013; Lewiner et al. 2003]. However, none of these methods is able to recover sharp features.

To capture sharp features of the iso-surface, Kobbelt et al. [2001] first introduced an extended marching cubes method (EMC) to insert additional feature points, given that the normals of some intersecting points are provided. Dual contouring (DC) [Ju et al. 2002] adapted this idea with Hermite data (the gradient of the implicit surface function). They proposed to insert one dual feature point inside a cube and then connect the dual points to form an iso-surface. DC does not need to perform the edge-flip operations required by [Kobbelt et al. 2001], but often generates non-manifold surfaces with many self-intersections. The non-manifold issue was later addressed in [Ju and Udeshi 2006], while the self-intersection issue was addressed in [Schaefer et al. 2007]. However, none of these two methods solves both the non-manifold and self-intersection problems simultaneously. Dual Marching Cubes (DMC) [Schaefer and Warren 2004] considers that the dual grid aligns with features of the implicit function, and extracts the iso-surface from the dual grid. DMC can preserve sharp features without excessive grid subdivisions as required by DC. Unfortunately, DMC still does not guarantee the generated mesh is self-intersection-free. Manson and



(a) $\boldsymbol{M}_i$ (36779, 0) (b) $\boldsymbol{M}_d$ (761821, 1076) (c) $\boldsymbol{M}_{\text{int}}$ (23805, 1088) (d) $\boldsymbol{M}_{\text{int}}$ (9433, 706) (e) $\boldsymbol{M}_o$ (2703, 390)

Fig. 2. The pipeline of our algorithm. The notation $(\bullet, \bullet)$ represents (number of faces, light field distance to the input mesh), where the latter is a measure of visual similarity between two 3D shapes, introduced by Chen et al. [2003]. (a): The input high-poly surface, which is not necessarily manifold, orientable, or self-intersection free. In this example, the input surface $\boldsymbol{M}_i$ is not 2-manifold, with 114 non-2-manifold edges, has 751 components and is not orientable (the back faces are rendered in black). It has over 36k faces, among which 23345 faces are self-intersected. (b): The extracted iso-surface $\boldsymbol{M}_d$ with $d = \frac{l}{200}$, where l is the bounding box diagonal size of $\boldsymbol{M}_i$ (Section 3.1). It is an orientable, water-tight, self-intersection free mesh with 19-components and 761k faces. (c-e) Our mesh optimization step (Section 3.2), during which we apply mesh simplification, flow, and alignment. While the simplification step may result in a slight increase in LFD, the subsequent flow and alignment steps enhance visual similarity. Consequently, the overall optimization step progressively reduces the light field distance and simplifies the mesh, and the intermediate meshes denoted as $\boldsymbol{M}_{\text{int}}$. The output $\boldsymbol{M}_o$ has only 2703 faces. Our approach resolves the existing topologic (non-manifoldness) and geometric issues (self-intersection), and approximates the high-poly with 7.3% faces.

Schaefer [2010] avoided self-intersections by subdividing each cube into multiple tetrahedra, and then applying marching tetrahedra (MT) to extract the iso-surface [Doi and Koide 1991]. This approach solves the self-intersection problems in the DMC approach, but the division of multiple tetrahedra, together with the employed octree structure, makes the algorithm either generate an overly dense mesh or require trial-and-error for suitable octree depth parameter settings. A survey about these approaches can be partially found in [de Araújo et al. 2015]. Recently, Portaneri et al. [2022] proposed an algorithm to generate watertight and orientable surfaces that strictly enclose the input. Their output is obtained by refining and carving the 3D Delaunay triangulation of the offset surface, however, still without the feature-preserving property.

There are also several learning-based approaches for iso-surface extraction. Deep marching cubes [Liao et al. 2018] and deep marching tetrahedra [Shen et al. 2021] learn differentiable MC and MT results. However, none of them can capture the sharp features of the initial surface. Neural marching cubes [Chen and Zhang 2021] and Neural dual contouring [Chen et al. 2022a] train the network to capture the sharp features without requiring extra Hermite information. However, the former generates self-intersected meshes, and the latter leads to non-manifold results. In Table 1, we summarize these methods and show their strength and weakness in terms of topologic and geometric properties: manifoldness, self-intersection-free, and sharp feature preservation.

3 METHOD

Given the input of a polygonal mesh $\boldsymbol{M}_i$, a maximum number of screen size n_p (i.e. the number of pixels covered by the diagonal length of the input's bounding box), and an optionally user-specified target number of triangles n_F, we seek to generate a triangle mesh $\boldsymbol{M}_o$ with the following properties:

[2]Although the initial paper results in non-manifold edges, this artifact was fixed by the follow-up works [Custodio et al. 2013; Lopes and Brodlie 2003]

Table 1. A Brief summary of the existing methods by their capabilities of maintaining geometry and topology properties. A more detailed survey can be found in [de Araújo et al. 2015].

Method	Manifold	Free of Self-Intersection	Preserve Features
Lorensen and Cline [1987]	√	√	×
Wyvill et al. [1986]	√	√	×
Chernyaev [1995]	√[2]	√	×
Doi and Koide [1991]	√	√	×
Kobbelt et al. [2001]	√	×	√
Ju et al. [2002]	×	×	√
Ju and Udeshi [2006]	√	×	√
Schaefer et al. [2007]	×	√	√
Manson and Schaefer [2010]	√	√	√
Portaneri et al. [2022]	√	√	×
Liao et al. [2018]	√	×	×
Shen et al. [2021]	√	×	×
Chen and Zhang [2021]	√	×	√
Chen et al. [2022a]	×	√	√

Pro.I The number of triangles in $\boldsymbol{M}_o$ is either minimized or equals to n_F if provided as a parameter;

Pro.II $\boldsymbol{M}_o$ is indistinguishable from $\boldsymbol{M}_i$ when rendered from a far-away view (a view where the diagonal length of the bounding box of $\boldsymbol{M}_i$ is less than n_p pixels);

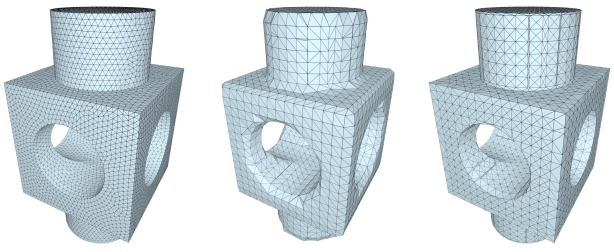
Pro.III $\boldsymbol{M}_o$ is both topologically and geometrically clean, i.e., water-tight, manifold, and intersection-free.

These three properties of $\boldsymbol{M}_o$ ensure rendering quality and enable any downstream geometric processing on it to have high computational efficiency, requiring no mesh repairing steps. The level of visual preservation in our second property is measured by Silhouette difference and the normal difference between $\boldsymbol{M}_i$ and $\boldsymbol{M}_o$. A similar normal indicates $\boldsymbol{M}_o$ preserves the sharp features of $\boldsymbol{M}_i$ as much as possible.

We follow several principles to design our approach: 1) We make no assumptions on the topologic or geometric properties of the input, allowing our approach to handle any models created in the wild; 2) We adopt an interior-point optimization-like strategy to realize the topology and geometry properties of Pro.III one by one: once a property is satisfied, it will be maintained for the rest of the steps; 3) We value robustness with the highest priority, so that our approach can process any inputs created by different domains of applications. Under guaranteed robustness, we further attempt to improve the computational efficacy to the greatest extent possible.

Overview. We tackle this problem in two main stages (Fig. 2), namely, *mesh extraction* (Section 3.1), and *mesh optimization* (Section 3.2). During the mesh extraction stage, we first compute an unsigned distance field for $\boldsymbol{M}_i$, then introduce a novel iso-surface mesh extraction approach for a positive offset distance d ($d = l/n_p$ as mentioned in Section 1), and finally remove all invisible disconnected components from the extracted iso-surface to obtain a mesh $\boldsymbol{M}_d$. Our generated $\boldsymbol{M}_d$ optimally recovers the sharp features implied by the d−iso-surface of the distance field, and guarantees watertightness, manifoldness, and free of self-intersections. The purpose of this stage is to generate a "clean" proxy mesh $\boldsymbol{M}_d$ of the input $\boldsymbol{M}_i$ that possibly has "dirty" topologic and geometric configurations. Our second mesh optimization stage involves a while-loop of three sequential steps: simplification, flow, and alignment. The simplification step aims to reduce the number of triangles of $\boldsymbol{M}_d$ by performing one pass of quadric edge-collapse decimation for the entire mesh; the flow step aims to pull $\boldsymbol{M}_d$ close to $\boldsymbol{M}_i$ via a per-vertex distance minimization; and the alignment step aims to optimize the surface normal of $\boldsymbol{M}_d$ so that the sharp features are maintained, which is achieved through local surface patch optimization. When the while loop stops, we output the final mesh $\boldsymbol{M}_o$. Since all three steps contain only local operations, the guarantees of $\boldsymbol{M}_d$ achieved during the first stage can be easily maintained by rolling back or skipping any operations that violate a guarantee. In the following sections, we provide technical details for each stage.

3.1 Mesh Extraction



(a) Input (b) MC33 (c) Ours

Fig. 3. The iso-surface meshes of a CAD model "block". Compared with classic MC33 algorithm [Chernyaev 1995], our iso-surfacing approach achieves a better visual similarity by recovering sharp features.

Given $\boldsymbol{M}_i$ and d, our goal is to extract an d−iso-surface mesh $\boldsymbol{M}_d$ that is watertight, manifold, feature-preserving, and self-intersection-free. Ensuring all these properties simultaneously is a challenging task. As an example, simply applying the well-known algorithm of MC33 cannot capture sharp features of the iso-surface as shown in Fig. 3. We tackle the mesh extraction problem by re-meshing the extracted local surface patches from templates of MC33. We selectively insert additional points to refine these local surface patches. Our key technique lies in the proposed mesh refinement technique that 1) guarantees topologically watertight and manifold properties and 2) captures geometric sharp features without causing self-intersection. We first compute a proper *discretization* of an unsigned distance function defined for $\boldsymbol{M}_i$, then analyze the connectivity changes when inserting new points to the MC33 templates to maintain the *topology guarantees* of the resulting mesh. After that, we focus on the *geometry fidelity* of the resulting mesh, i.e. feature-preserving and self-intersection-free. For the extracted mesh $\boldsymbol{M}_d$, we finally remove invisible components.

Discretization. An unsigned distance field is defined as a function:

$$f(\boldsymbol{p}) := \min_{\boldsymbol{q} \in \boldsymbol{M}_i} \|\boldsymbol{p} - \boldsymbol{q}\|, \tag{1}$$

where $\boldsymbol{p} \in \mathcal{R}^3$. The implicit function of d-iso-surface is $f(\boldsymbol{p}) = d$. Since the explicit representation of the d-iso-surface is intractable, we follow the general pipeline of prior iso-surfacing approaches that first voxelize the ambient space around $\boldsymbol{M}_i$, and then approximate the solution through extracting local surface patches within each voxel. Since the local patches are typically simple, the voxel size plays an important role in the to-be-extracted mesh. A coarse voxel size can miss important solutions, such as the one illustrated in Fig. 4, where no d-iso-surface could be extracted if a grid size large than $2d$ is used, while an excessively small voxel size will result in a dense grid that is time-consuming to compute (Fig. 15). By default, we choose the edge length of a voxel to be $d/\sqrt{3}$ to avoid geometric feature losses as illustrated in the Fig. 4.

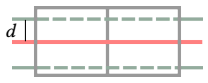
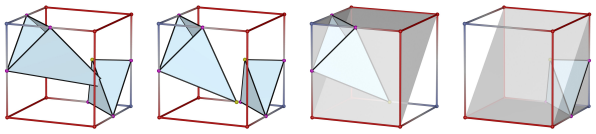


Fig. 4. The d-iso-surface (green lines) of the input mesh (red line) cannot be captured if the voxel size is larger than $2d$.

Topologic Guarantees. For each voxel, we employ existing templates to decide the iso-contours [Chernyaev 1995; Custodio et al. 2013; Lorensen and Cline 1987], and then insert an additional point for each contour. The templates of either the original MC [Lorensen and Cline 1987] or MC33 [Chernyaev 1995; Custodio et al. 2013] can be used to generate the iso-contours since they both ensure the extracted mesh is watertight and manifold. We choose MC33 in this work since it covers more linear interpolation cases and resolves the ambiguity in MC, thus extracting iso-surface meshes with generally smaller genus [Chernyaev 1995; Custodio et al. 2013]. As illustrated in Fig. 23, we insert one vertex per iso-contour surface if it is homemorphic to a disk, where the iso-contour surfaces of cases 4.1.2, 6.1.2, 7.4.2, 10.1.2 and 12.1.2, and one iso-contour of case 13.5.2 are excluded since they have two boundaries. This scheme



(a) Without Constraints (b) With Constraints (c) Polyhedron$_0$ (d) Polyhedron$_1$

Fig. 5. Illustration of our feature-aware iso-surface extraction step for MC33 case 4.1.1. The cube vertices with iso-value smaller than d is marked as blue, while the red vertices are the opposite. The iso-points are marked as pink, and the feature points are marked as yellow. Without forcing the feature points within their belonging polyhedra (c, d), it is easy to obtain a mesh with self-intersections (a).

ensures that the additional vertices neither bring any non-manifold configurations nor create holes in the resulting mesh.

Feature Vertex Insertion. We use one vertex per local patch with a disk-topology to provide more degrees of freedom to capture sharp features. Its position can be computed by minimizing the distance to patch vertices along the normal directions:

$$\arg\min_{\boldsymbol{x}} \sum_i \left(\boldsymbol{n}_{\boldsymbol{p}_i} \cdot (\boldsymbol{x} - \boldsymbol{p}_i)\right)^2, \tag{2}$$

where $\boldsymbol{x}$ is the desired position, $\boldsymbol{p}_i$ and $\boldsymbol{n}_{\boldsymbol{p}_i}$ denote an iso-contour vertex and its normal, respectively, and the summation goes through all patch vertices. However, without any constraints, $\boldsymbol{x}$ may be arbitrarily positioned and cause surface intersections in the extracted mesh. This issue is deteriorated by the template cases with multiple iso-contours. Fig. 5 illustrates such as example using MC33 case 4.1.1.

We propose a simple approach to recover feature vertices without mesh intersections. Given a voxel, we subdivide it into convex polyhedra within which the feature vertices are constrained. As illustrated in Fig. 24, the subdivision is performed according to the number and the different configurations of the iso-contours with a disk-topology. For example, for cases with only a single disk-topology iso-contour, such as case 1, no subdivision is involved and the polyhedron is the voxel itself, and for those with multiple iso-contours, such as case 4.1.1, convex and planar polygons are needed to partition the voxel into non-overlapping polyhedra. If we constrain the position of the inserted vertex stays inside its belonging polyhedron, the extracted mesh is guaranteed to incur no self-intersections.

Accordingly, for each inserted vertex, we obtain its coordinate x by solving a linear constrained quadratic programming:

$$\begin{aligned} \arg\min_{\boldsymbol{x}} & \sum_i \left(\boldsymbol{n}_{\boldsymbol{p}_i} \cdot (\boldsymbol{x} - \boldsymbol{p}_i)\right)^2 \\ s.t. \quad & \boldsymbol{n}_s \cdot (\boldsymbol{x} - c_f) < 0, \ \forall \boldsymbol{n}_s \in N_s \end{aligned} \tag{3}$$

where N_s is the set of face normals of the corresponding polyhedron of x, and c_f is the center of the corresponding polyhedron face. To compute $\boldsymbol{n}_{\boldsymbol{p}_i}$, if $\boldsymbol{p}_i \in \boldsymbol{M}_i$, we simply use the mesh normal, otherwise, we first solve for any $\boldsymbol{p}_i^\star \in \arg\min_{\boldsymbol{p} \in \boldsymbol{M}_i} \|\boldsymbol{p}_i - \boldsymbol{p}\|$ and then let $\boldsymbol{n}_{\boldsymbol{p}_i} := (\boldsymbol{p}_i - \boldsymbol{p}_i^\star)/\|\boldsymbol{p}_i - \boldsymbol{p}_i^\star\|$.

Feature Extraction. The previous step recovers vertices on sharp features of the d-iso-surface. But their connections may not align well with the sharp edges, the highlighted region in Fig. 6b demonstrates this issue. Furthermore, since the sharp features exist in a small fraction of voxels, we aim at a minimal increase in the additional feature edges and vertices by inserting only those feature vertices on sharp features and using the original MC33 templates as much as possible. However, we do not know the sharp features of the d-iso-surface as prior. Therefore, we introduce a posterior approach to recover the necessary feature curves, which involves two phases: *Feature Edge Adjustment*, and *Feature Filtering*. The first phase generates an iso-surface mesh by considering all inserted feature vertices as sharp features. With this iso-surface mesh, we can use existing automatic feature identification approaches to obtain sharp features. The second phase generates the actual iso-surface mesh $\boldsymbol{M}_d$ by blending the iso-contour patches containing the detected feature vertices with those original MC33 patches that do not contain any sharp features.

Feature Edge Adjustment. During the first phase, we insert a feature vertex for every disk-topologic iso-contour patch. We then perform an edge-flip operation for every mesh edge if the flipped edge connects two inserted feature vertices (see Algorithm 2 for more details). To ensure the self-intersection-free guarantee, we skip those edge-flips that may cause self-intersections. This phase can already produce an iso-surface mesh that satisfies the desired topologic and geometric properties. However, as mentioned earlier, this iso-surface mesh contains more elements than desired and those unnecessary "fake" sharp features are noisy and not visual-appealing (see the top two zoomed-in figures in Fig. 6c).

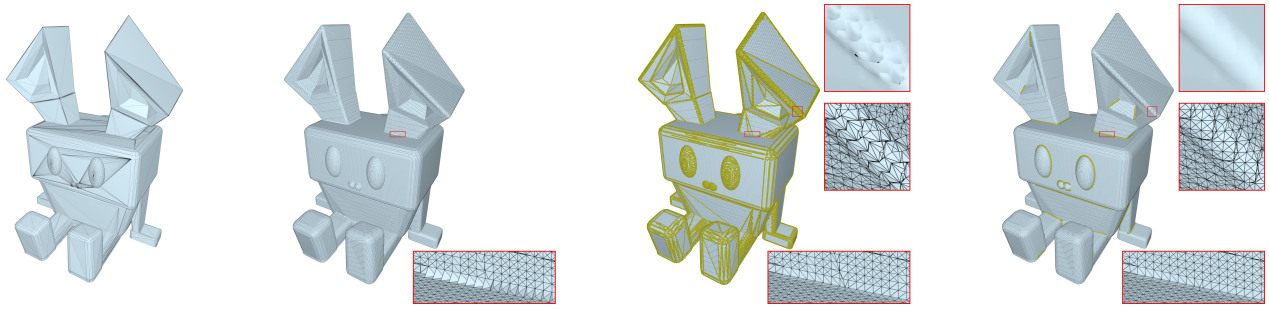
Feature Filtering. During the second phase, we first extract a feature graph from the resulting mesh of the first phase. The feature graph is composed of a set of feature curves where each curve is a sequence of mesh edges with its dihedral angle smaller than θ_0 (see [Gao et al. 2019] for details). We then mark a feature curve as valid if it is composed of more than l_0 mesh edges. The valid feature curves are considered to contain "real" sharp features to recover. After that, for each iso-contour patch, we keep those inserted feature vertices if they are on the valid feature curves, otherwise, we use their original template. This step removes a lot of noisy, "fake" feature edges. Finally, we perform the edge flip algorithm Algorithm 2 once more to extract the d-iso-surface mesh $\boldsymbol{M}_d$.

Our feature recovery algorithm performs well for the models with various features that can be represented by piecewise line segments, e.g. sharp curves in Fig. 3 and the eye and beak contours in Fig. 12.

Interior Removal. Since we use an unsigned distance function, our final extracted iso-surface $\boldsymbol{M}_d$ may have interior components, which are totally invisible. Given the generated mesh $\boldsymbol{M}_d$ are watertight and free of self-intersection, we can apply the in-and-out test and remove the components which is purely inside of any of the others.

3.2 Mesh Optimization

Starting from a mesh $\boldsymbol{M}_d$ that is watertight, manifold, feature preserving, and self-intersection-free, we now introduce an iterative mesh optimization approach to obtain a final $\boldsymbol{M}_o$ that satisfies our



(a) Input mesh $\boldsymbol{M}_i$ (b) Unflipped feature mesh (c) Flipped feature mesh (e) Final iso-surface $\boldsymbol{M}_d$

Fig. 6. Our iso-surface extraction pipeline. We mark the initial feature points in the third figure and the remaining ones after applying feature graph filter in the last figure (the yellow points). Our post process successfully resolves the sawtooth artifacts around the ear of the character (the top two zoomed-in figures in the third and last columns, with the top ones are rendered without wireframes), but still keeps the major sharp features, for example, the bottom zoomed-in figures.

Algorithm 1 Iso-surface Extraction

Input: $\boldsymbol{M}_i$, d, θ_0, l_0
Output: $\boldsymbol{M}_d$
$G \leftarrow$ **gridDiscretization**$(\boldsymbol{M}_i, d)$ ▷ generate the grids
Compute $f(\boldsymbol{p})$ for all grid points $\boldsymbol{p}$ in G ▷ Equation 1
for each *cube* $\in G$ **do** ▷ iso-surface extraction
  Lookup for the template case ▷ [Chernyaev 1995], Fig. 23
  for each Disk-topologic patch in cube case **do**
    Compute the iso-points on cube edges
    Form the quadratic program ▷ Equation 3, Fig. 24
    Solve for feature vertices ▷ Fig. 23
  end for
end for
$\boldsymbol{M}_d \leftarrow$ **edgeFlip**$(\boldsymbol{M}_d)$ to connect feature vertices ▷ Algorithm 2
$\boldsymbol{M}_d \leftarrow$ **featureFilter**$(\boldsymbol{M}_d, \theta, l_0)$
$\boldsymbol{M}_d \leftarrow$ **removeInterior**$(\boldsymbol{M}_d)$

Algorithm 2 Edge Flip

Input: $\boldsymbol{M}_d$
Output: $\boldsymbol{M}_d$ ▷ mesh after edge-flips
$Q \leftarrow \{\}$
BVHTree T
T.build$(\boldsymbol{M}_d)$ ▷ [Karras 2012]
for each edge $e \in \boldsymbol{M}_d$ **do**
  if e.**oppVs** are feature vertices **then**
    Q.**push**(e) ▷ opposite vertices are feature vertices
  end if
end for
while $Q \neq \{\}$ **do**
  $e \leftarrow Q$.**top**()
  if e was not flipped before **then**
    if isIntersectionFree$(\boldsymbol{M}_d, \mathrm{T}, e)$ **then**
      flipEdge$(\boldsymbol{M}_d, e)$
      T.refit$(\boldsymbol{M}_d)$ ▷ update BVH [Karras 2012]
    end if
  end if
end while

three desired properties, i.e., Pro.I-III. As shown in Algorithm 3, our optimization involves a maximum of N iterations of three sequential steps: *simplification*, *flow*, and *alignment*. We stop the iterations until either the Hausdorff distance between the simplified meshes of two consecutive loops (relative change) is smaller than ϵ, the loop number reaches N, or the target face number reduces below n_F, where the first condition has the highest priority by default. Each of the three steps involves only local operations. To ensure our optimization proceeds towards the generation of $\boldsymbol{M}_o$ with the desired properties, we perform the following checks for the meshes before and after applying a local operation:

(1) Topology consistency: the updated mesh is manifold, watertight, and has the same genus and the number of components as the mesh before applying the local operation;
(2) Self-intersection-free: the updated mesh is free of intersections.

Mesh Simplification. This step aims to achieve the first property of $\boldsymbol{M}_o$, i.e., $\boldsymbol{M}_o$ contains as few triangles as possible. We perform an entire pass of the standard edge-collapse operation for all edges of $\boldsymbol{M}_o$ to reduce as many faces as possible or match the target face number n_F, where the coordinate of the newly generated vertices are determined by the quadratic edge metric (QEM) [Garland and Heckbert 1997] weighted by virtual planes for each edge to avoid the degeneracy in planar regions. Importantly, the topologic and geometric validity of $\boldsymbol{M}_o$ is maintained during the simplification process by skipping those edge-collapse operations that may violate the aforementioned checks. Moreover, to ensure we get closer to $\boldsymbol{M}_i$, we also skip the collapse operations which increase the distance between affected local triangle patches and $\boldsymbol{M}_i$. For efficiency concerns, we only compute the one-sided distance from the local patch to the input mesh. This one-sided check may result in the acceptance

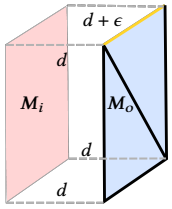


Fig. 7. Collapsing the yellow edge reduces the distance from $\boldsymbol{M}_o$ to $\boldsymbol{M}_i$ ($d+\epsilon \rightarrow d$), but leads to an undesirable visual appearance.

of unexpected collapses, as illustrated in Fig. 7. To overcome this, we further skip the operations leading to a Hausdorff distance larger than d, where the two involving meshes are the ones before and after the local operation and the Hausdorff distance is computed approximately by sampling points on the local triangle patches as in [Cignoni et al. 1998]. We show the comparison in Fig. 8. Without the guarantee of a distance decrease, we will lose some important information. Without the guarantee of a small Hausdorff distance, we may end up with larger silhouette difference and normal difference, that is, worse visual similarity.

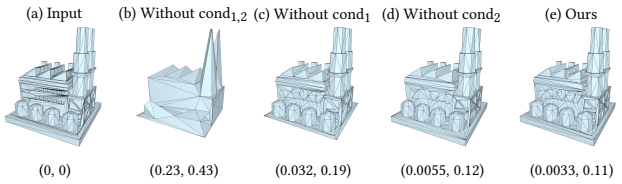


Fig. 8. The results of different simplification conditions. $(\bullet, \bullet)$ denotes (silhouette difference, normal difference). cond$_1$: skip the collapse which increases the vertex-surface distance to $\boldsymbol{M}_i$; cond$_2$: skip the collapse which results a large Hausdorff distance. After applying the both conditions, we achieve a better visual score.

Mesh Flow. Our mesh flow step brings $\boldsymbol{M}_o$ geometrically close to $\boldsymbol{M}_i$ and reduces the silhouette visual differences between the two meshes. The detailed algorithm is provided in Algorithm 3 (Line 7-11).

Algorithm 3 Mesh Optimization Process

Input: $\boldsymbol{M}_i$, $\boldsymbol{M}_d$, d, n_F N, r, ϵ
Output: $\boldsymbol{M}_o$
1: $\boldsymbol{M}_o \leftarrow \boldsymbol{M}_d$
2: $l \leftarrow$ **bboxSize**$(\boldsymbol{M}_i)$ ▷ bounding box diagonal size
3: **for** $i = 0$ to N **do**
4: $\quad \boldsymbol{M}' \leftarrow \boldsymbol{M}_o$
5: $\quad \boldsymbol{M}_o \leftarrow$ **meshSimplification**$(\boldsymbol{M}_i, \boldsymbol{M}_o, d, n_F)$ ▷ Algorithm 5
6: $\quad \widetilde{\boldsymbol{M}} \leftarrow \boldsymbol{M}_o$
7: $\quad$ **for each** vertex $\boldsymbol{v} \in \boldsymbol{M}_o$ **do** ▷ mesh flow step
8: $\quad\quad \boldsymbol{v}^* \leftarrow \mathbf{argmin}_{\boldsymbol{u} \in \boldsymbol{M}_i} \|\boldsymbol{u} - \boldsymbol{v}\|$
9: $\quad\quad d_{\boldsymbol{v}} \leftarrow r(\boldsymbol{v}^* - \boldsymbol{v})$ ▷ successive flow, $r < 1$
10: $\quad\quad \boldsymbol{v} \leftarrow$ **localUpdate**$(\boldsymbol{M}_o, \boldsymbol{v}, d_{\boldsymbol{v}})$ ▷ Algorithm 4
11: $\quad$ **end for**
12: $\quad$ **for each** vertex $\boldsymbol{v} \in \boldsymbol{M}_o$ **do** ▷ feature alignment step
13: $\quad\quad \boldsymbol{v}_{\text{opt}} \leftarrow$ **featureAlignment**$(\widetilde{\boldsymbol{M}}, \boldsymbol{M}_o, \boldsymbol{v})$ ▷ Algorithm 6
14: $\quad\quad d_{\boldsymbol{v}} \leftarrow \boldsymbol{v}_{\text{opt}} - \boldsymbol{v}$
15: $\quad\quad \boldsymbol{v} \leftarrow$ **localUpdate**$(\boldsymbol{M}_o, \boldsymbol{v}, d_{\boldsymbol{v}})$ ▷ Algorithm 4
16: $\quad$ **end for**
17: $\quad$ **if Hausdorff**$(\boldsymbol{M}_o, \boldsymbol{M}') < \epsilon \cdot l$ **then**
18: $\quad\quad$ **Break** ▷ update is small enough
19: $\quad$ **end if**
20: **end for**

When actually applying the mesh flow process, for each vertex $\boldsymbol{v}$ in $\boldsymbol{M}_o$, we find its Euclidean-distance-wise closest point $\boldsymbol{v}^*$ of $\boldsymbol{M}_i$ and successively push $\boldsymbol{v}$ to $\boldsymbol{v}^*$ along the vector $d_{\boldsymbol{v}} = \boldsymbol{v}^* - \boldsymbol{v}$. Instead of updating $\boldsymbol{v}$ to $\boldsymbol{v}^*$ directly, we deform $\boldsymbol{v}$ towards $\boldsymbol{v}^*$ based on a constant fractional ratio r of the vector, which allows more moving space for the entire mesh and reduces the chance of optimization stuck when $\boldsymbol{M}_o$ is still far from $\boldsymbol{M}_i$. We also apply a simple line search for the self-intersection-free check to find the maximum step size during the local deformation (Algorithm 4).

Algorithm 4 Local Update

Input: $\boldsymbol{M}$, $\boldsymbol{v}$, $d_{\boldsymbol{v}}$
Output: updated $\boldsymbol{v}$
Note: $\boldsymbol{v}$ is a vertex of $\boldsymbol{M}$.
1: $\alpha \leftarrow 1$
2: **while** $\boldsymbol{v} + \alpha d_{\boldsymbol{v}}$ leads to self-intersections **do**
3: $\quad \alpha \leftarrow \alpha/2$
4: **end while**
5: **return** $\boldsymbol{v} + 0.95\alpha d_{\boldsymbol{v}}$ ▷ Using 0.95 to avoid numerical error

Algorithm 5 Mesh Simplification

Input: $\boldsymbol{M}_i$, $\boldsymbol{M}_o$, d, n_F
Output: simplified mesh $\boldsymbol{M}_o$
Notes: $\boldsymbol{M}_i$ is the reference mesh, n_F is optional
1: Form priority queue Q ▷ [Garland and Heckbert 1997]
2: **BVHTree** T
3: T.build$(\boldsymbol{M}_o)$ ▷ [Karras 2012]
4: **while** $Q \neq \{\}$ **do**
5: $\quad e \leftarrow Q$.**top**()
6: $\quad$ **if** e has been visited before **then**
7: $\quad\quad$ **continue** ▷ has been collapsed
8: $\quad$ **end if**
9: $\quad$ **if topologyConsistencyCheck**$(\boldsymbol{M}_o, e)$ failed **then**
10: $\quad\quad$ **continue** ▷ [Cignoni et al. 2008]
11: $\quad$ **end if**
12: $\quad$ **if not isIntersectionFree**$(\boldsymbol{M}_o, \text{T}, e)$ **then**
13: $\quad\quad$ **continue** ▷ collapse will introduce intersections
14: $\quad$ **end if**
15: $\quad \boldsymbol{M}_e \leftarrow$ sub-mesh of $\boldsymbol{M}_o$ adjacent to e
16: $\quad \boldsymbol{M}'_e \leftarrow \boldsymbol{M}_e$ after collapse
17: $\quad$ **if dist**$(\boldsymbol{M}_e \rightarrow \boldsymbol{M}_i) <$ **dist**$(\boldsymbol{M}'_e \rightarrow \boldsymbol{M}_i)$ **then**
18: $\quad\quad$ **continue** ▷ collapse increases the distance to $\boldsymbol{M}_i$
19: $\quad$ **end if**
20: $\quad$ **if dist**$(\boldsymbol{M}_e \rightarrow \boldsymbol{M}'_e) > d$ or **dist**$(\boldsymbol{M}'_e \rightarrow \boldsymbol{M}_e) > d$ **then**
21: $\quad\quad$ **continue** ▷ distance update is too large
22: $\quad$ **end if**
23: $\quad$ **collapseEdge**$(\boldsymbol{M}_o, e)$ ▷ satisfy all desired properties
24: $\quad$ T.refit$(\boldsymbol{M}_o)$ ▷ update BVH [Karras 2012]
25: $\quad$ **if** n_F is given and $\boldsymbol{M}_o$.**faceNumber** $\leq n_F$ **then**
26: $\quad\quad$ **break**
27: $\quad$ **end if**
28: **end while**
29: **return** $\boldsymbol{M}_o$

Algorithm 6 Feature Alignment

Input: $\widetilde{M}$, M_o, v
Output: updated v which preserves features
Require: $\widetilde{M}$ and M_o has the same mesh connectivity
1: **for each** $f \in N^1(v)$ **do** ▷ loop over adjacent faces
2: $n_f \leftarrow e_0 \times e_1$ ▷ unnormalized face normal of M_o
3: $c_n \leftarrow \|n_f\|$ ▷ get the initial norm
4: $\tilde{n}_f \leftarrow \tilde{e}_0 \times \tilde{e}_1$ ▷ unnormalized face normal of $\widetilde{M}$
5: **end for**
6: Fix c_n, get v_{opt} by minimizing Equation 5 ▷ quadratic program
7: **return** v_{opt}

Feature Alignment. The previous mesh flow can stretch the mesh unanimously, breaking features and creating dirty inputs for subsequent mesh simplification and flow procedure (see Fig. 9 for an example). We thus introduce a feature alignment step. For each vertex v, we seek an optimized position v_{opt} by minimizing the shape difference between the local surface of v_{opt} and that of v before mesh flow:

$$E(v) := \sum_{f \in \mathbf{N}^1(v)} \left\| \frac{n_f}{\|n_f\|} - \frac{\tilde{n}_f}{\|\tilde{n}_f\|} \right\|^2, \tag{4}$$

where we use the normal disagreement to approximate the shape difference (Line 6 in Algorithm 6), $\mathbf{N}^1(v)$ is the faces within the 1-ring neighbor of v, and n_f, $\tilde{n}_f$ are the unnormalized face normal of the current mesh and the one before the flow respectively. The summation takes over all faces within the 1-ring neighborhood of vertex v. This face normal difference summation approximates the vertex normal difference. Notice that Equation 4 is a nonlinear function, which can be solved by the classical Newton's Method. In order to improve the efficiency, we instead treat the $\|n_f\|$ as constant (equal to the value at the beginning of the alignment step, denoted as c_n) and solve a quadratic approximation of Equation 4:

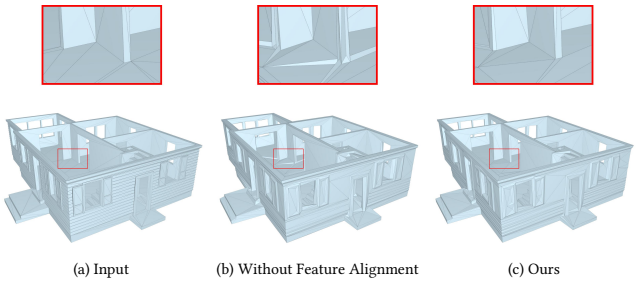
$$E(v) := \sum_{f \in N^1(v)} \left\| \frac{n_f}{c_n} - \frac{\tilde{n}_f}{\|\tilde{n}_f\|} \right\|^2. \tag{5}$$

Once we obtain the corresponding v_{opt} that minimizes Equation 5, we update v to be v_{opt} with the line search Algorithm 4 to prevent self-intersections. Given this local operation only slightly updates the mesh, our quadratic approximation leads to small errors, but in turn, significantly boosts performance (turning a non-convex problem to an unconstrained quadratic program).

3.3 Self-intersection Check Acceleration

Starting from an intersection-free 3D triangle mesh, our low-poly re-meshing pipeline could introduce intersections when performing the edge flips during mesh extraction, the edge collapses during mesh simplification, and the vertex optimization during the mesh flow and the feature alignment steps. Note that we say a mesh has intersections when any of its two triangles overlaps, or any of its two non-adjacent triangles touch or intersect.

Before discussing our accelerated check of self-intersections, we first introduce the necessary notations. For a vertex v, we denote



(a) Input (b) Without Feature Alignment (c) Ours

Fig. 9. Without feature alignment step, we will end up with the results with "spikes" (see zoomed-in region for details).

$\mathbf{N}^i(v)$ as the set of all triangles that are bounded within its i-th ring neighborhood. For example, for the bottom left image in Fig. 10, $\mathbf{N}^1(v)$ is the red region, and $\mathbf{N}^2(v)$ is the union of red and green region[3]. We further denote M_e as the local neighborhood related to a certain local operation, where M_e endows different definitions. For edge flip, we define $M_e = \{f_1, f_2\}$ where f_1 and f_2 are the two neighboring triangles of e. For edge collapse, $M_e = \mathbf{N}^1(v')$ where v' is the newly created vertex. For vertex optimization (otherwise known as smoothing), we let $M_e = \mathbf{N}^1(v)$. Moreover, we let M_s be the sub-mesh formed by all faces that share at least one vertex with M_e but not in M_e (the green regions in Fig. 10). Finally, we let $M_1 = M_s \bigcup M_e$. The rest of the mesh are denoted as M_r (the blue regions in Fig. 10).

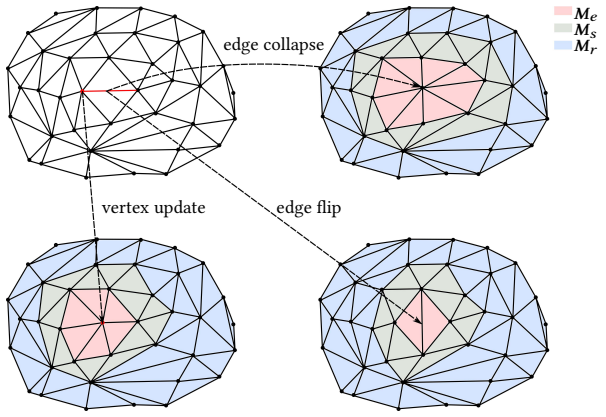


Fig. 10. Edge or vertex operation illustration. $M_1 = M_e \bigcup M_s$.

We note that a mesh-reduction operation does not introduce self-intersection iff the following two conditions hold:

(1) M_e does not intersect M_r;
(2) M_1 is self-intersection free.

Here we skip the intersection check within M_s, M_r and between M_s and M_r, because M_s and M_r are the unchanged sub-mesh of the mesh before the local operation, which is free of self-intersections. In general, the two conditions above can be check by conventional

[3]These sets of triangles are called "topological neighborhoods", introduced in [Attene 2010]

Table 2. A speedup summary of self-intersection checks (used in edge flip, edge collapse, and vertex optimization steps) for some of the figures shown in the paper. The upper index of the figures indicates the corresponding row of that figure. T_n: the time cost of the neighboring triangle intersection check only using surface normal test. T'_n: the same time information with full normal cone test (surface normal + contour test). $T^\star_\mathrm{n}$: the same time information but applying parallel triangle pairs intersection check. $T_\mathrm{t}, T'_\mathrm{t}, T^\star_\mathrm{t}$: the total time information of the whole intersection-check process (neighboring triangle intersection check + BVH check), corresponding to $T_\mathrm{n}, T'_\mathrm{n}, T^\star_\mathrm{n}$.

Figures	T_n(s)	T'_n(s)	$T'_\mathrm{n}/T_\mathrm{n}$	$T^\star_\mathrm{n}$(s)	$T^\star_\mathrm{n}/T_\mathrm{n}$	T_t(s)	T'_t(s)	$T'_\mathrm{t}/T_\mathrm{t}$	$T^\star_\mathrm{t}$(s)	$T^\star_\mathrm{t}/T_\mathrm{t}$
Fig. 2	23.80	2587.09	108.71	111.59	4.69	119.31	2689.11	22.54	213.63	1.79
Fig. 6	16.15	2847.69	176.31	119.82	7.42	115.86	2955.23	25.51	224.50	1.94
Fig. 11(d)^0	21.98	2653.21	120.70	158.18	7.20	120.35	2769.56	23.01	266.17	2.21
Fig. 12(g)^0	13.51	2409.66	178.40	70.66	5.23	93.06	2497.08	26.83	156.08	1.68
Fig. 12(g)^1	10.24	1377.05	134.51	43.98	4.30	57.25	1421.92	24.84	87.36	1.53
Fig. 13(j)^0	58.93	7958.25	135.05	552.99	9.38	360.12	8291.89	23.03	885.49	2.46
Fig. 13(j)^1	5.92	819.00	138.37	39.56	6.68	29.79	846.21	28.41	65.04	2.18
Fig. 14(d)	13.92	2284.73	164.08	106.71	7.66	99.87	2382.30	23.86	200.90	2.01
Fig. 22(a)^2	15.65	2283.99	145.93	96.42	6.16	98.12	2376.82	24.22	187.28	1.91
Average	20.01	2802.30	144.67	144.43	4.69	121.52	2914.46	24.69	254.05	1.79

triangle-triangle intersection test. However, checking the first condition above is computationally inefficient, especially when $\boldsymbol{M}_r$ contains lots of triangles. Given $\boldsymbol{M}_e$ does not share any vertex or edge with $\boldsymbol{M}_r$, this part can be handled by standard BVH-based collision detection. The detailed algorithm is given in Algorithm 7.

Algorithm 7 BVH Meshes Intersection Check

Input: $\boldsymbol{M}_e$, $\boldsymbol{M}_r$, T (BVH tree of $\boldsymbol{M}_r$)
Output: whether $\boldsymbol{M}_e$ intersects with $\boldsymbol{M}_r$
Notes: all faces $\boldsymbol{M}_e$ do not share vertices with the faces in $\boldsymbol{M}_r$

```
for each face f ∈ M_e do
    f_1 ← T.closestFace(f)                ▷ get the closest face
    if triTriIntersection(f, f_1) then
        return true                       ▷ does intersect
    end if
end for
return false                              ▷ does not intersect
```

Unfortunately, for the second case, all the faces in $\boldsymbol{M}_e$ share at least one vertex with the faces in $\boldsymbol{M}_s$. The BVH-based acceleration is no longer efficient, as the shared features always lead to failure in BVH culling. In this scenario, the naive approach involves $|\boldsymbol{M}_e|\cdot|\boldsymbol{M}s|$ pairs of triangle-triangle intersection check. Although $|\boldsymbol{M}_e|$ and $|\boldsymbol{M}_s|$ are usually small for one local operation, the three edge flip, edge collapse, and vertex optimization operations will typically be executed for a massive number of times during the entire re-meshing pipeline. In practice, we find that this $\boldsymbol{M}_1$ intersection-free check takes $\sim$ 50% of the computational time of the whole intersection check process. Avoiding unnecessary triangle-triangle intersection checks, which is expensive to compute, will lead to a dramatic speedup. To this end, we note that $\boldsymbol{M}_1$ is open, and to check whether a mesh with boundaries has self-intersection, Volino and Thalmann [1994] introduce a theory providing a sufficient condition: Let $\boldsymbol{M}$ be a continuous surface, bounded by $\partial\boldsymbol{M}$, $\boldsymbol{M}$ is self-intersection free if there exists a vector $\boldsymbol{n}$, such that:

(1) *Surface Normal Test*: For every point $\boldsymbol{p} \in \boldsymbol{M}, \boldsymbol{n_p}\cdot\boldsymbol{n} > 0$, where $\boldsymbol{n_p}$ is the surface normal at $\boldsymbol{p}$;
(2) *Contour Test*: The projection of the contour $\partial\boldsymbol{M}$ along the $\boldsymbol{n}$ is not self-intersected.

They also provide a discrete version for triangle meshes:

(1) *Surface Normal Test*: The angle of the normal cone formed by all triangle face normals is less than $\frac{\pi}{2}$;
(2) *Contour Test*: The projection of the mesh boundary $\partial\boldsymbol{M}$ along the normal cone axis is not self-intersected.

For the first test, one can use the tight normal cone merging algorithm mentioned by Han et al [2021], and for the second test, Wang et al [2017] proposed a side-sign based unprojected contour test.

Surface normal test only need $|\boldsymbol{M}_1|$ times normal cone expansion [Han et al. 2021]. As shown in Table 2, only applying *Surface Normal Test* results in ~145× speedup compared with the full normal cone test, and 4.69× speed up compared with parallel triangle-triangle pair check. Moreover, this normal cone test acceleration speeds up the whole self-intersection check process by 24.69× and 1.79× compared with the full normal cone test and parallel triangle-triangle pair check, respectively. Surface normal test alone in practice is enough to generate a surface without self-intersection. We perform only the surface normal test during the self-intersection check, and if it fails, we apply direct triangle-triangle pair checks. Although the surface normal test alone is not sufficient to ensure free of self-intersection of $\boldsymbol{M}_1$, in practice, we find our final output $\boldsymbol{M}_o$ is always self-intersection free. We also perform a self-intersection check of $\boldsymbol{M}_o$. If $\boldsymbol{M}_o$ intersects itself, we remove the surface normal test filter, and re-run the algorithm with direct triangle-triangle pair checks.

4 EXPERIMENTS

We implement our algorithm in C++, using Eigen for linear algebra routines, CGAL [Brönnimann et al. 2022] for exact triangle-triangle intersection check, libigl [Jacobson et al. 2018] for basic geometry processing routines. We use the fast winding number [Barill et al. 2018] for interior components identification. We implement the bottom-up BVH traversal algorithm mentioned in [Karras 2012] to refit the BVH for self-intersection check, and use Metro [Cignoni et al. 1998] for Hausdorff distance computation. Unless particularly mentioned, we set $n_p = 200$, $\theta_0 = 120^\circ$, $l_0 = 4$, $N = 50$, $r = \frac{1}{8}$, and