DMesh++: An Efficient Differentiable Mesh for Complex Shapes

Sanghyun Son*
University of Maryland
shh1295@umd.edu

Matheus Gadelha
Adobe Research
gadelha@adobe.com

Yang Zhou
Adobe Research
yazhou@adobe.com

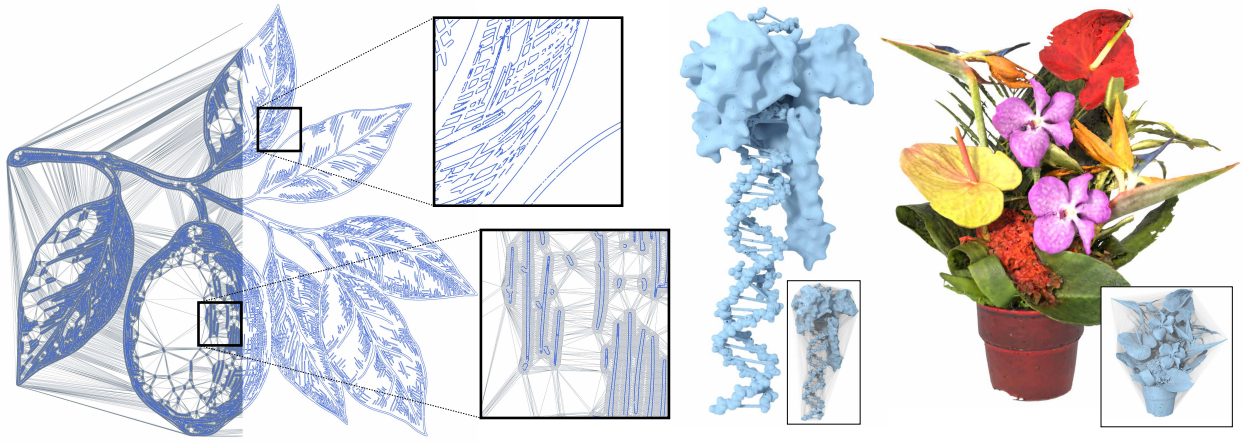
Matthew Fisher
Adobe Research
matfishe@adobe.com

Zexiang Xu
Adobe Research
zexu@adobe.com

Yi-Ling Qiao
University of Maryland
yilingq@umd.edu

Ming C. Lin
University of Maryland
lin@umd.edu

Yi Zhou
Adobe Research
yizho@adobe.com



(a) 2D Point Cloud Recon. (b) 3D Point Cloud Recon. (b) 3D Multi View Recon.

Figure 1. **DMesh++ for complex 2D and 3D shapes.** DMesh++ encodes all geometric and topological information into continuous point features. (a) By optimizing these point features, DMesh++ is able to reconstruct complex 2D drawings from sample points. (b) This approach is also applicable to 3D, where it reconstructs the complex geometric structure of DNA from a point cloud. (c) By incorporating additional color features, DMesh++ can reconstruct complex, colored 3D shapes from multi-view images. For each result, the *"imaginary"* part is rendered in gray, while the *"real"* part—which defines the final mesh—is rendered in other colors. (Sec. 3.1).

Abstract

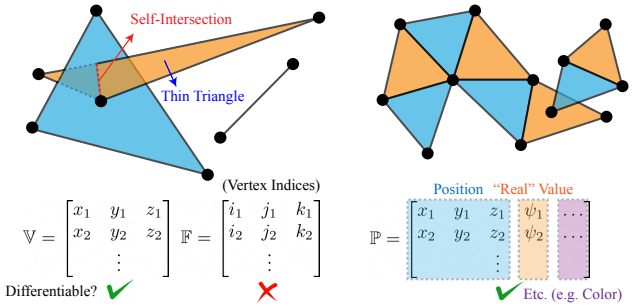
Recent probabilistic methods for 3D triangular meshes capture diverse shapes by differentiable mesh connectivity, but face high computational costs with increased shape details. We introduce a new differentiable mesh processing method that addresses this challenge and efficiently handles meshes with intricate structures. Our method reduces time complexity from $O(N)$ to $O(\log N)$ and requires significantly less memory than previous approaches. Building on this innovation, we present a reconstruction algorithm capable of generating complex 2D and 3D shapes from point clouds or multi-view images. Visit our project page for source code and supplementary material.

* This work was mainly done during internship at Adobe Research and continued as a collaborative effort with UMD.
** The paper was last modified on Jul. 6, 2025.

1. Introduction

Among various possible shape representations, a mesh is often favored for a wide range of downstream tasks due to its efficiency, versatility, and controllability. A mesh is defined by its vertices' position and their connectivity in the form of edges and faces. This connectivity is discrete in nature, and also the number of possible connectivities grows exponentially with the number of points, which prevents meshes from being differentiable shape representations (Fig. 2). To address this, recent data-driven efforts have attempted to predict mesh connectivity using Transformer-based models [5, 6, 40, 41]. However, these methods face inherent challenges with robustness to outlier meshes, potential self-intersections, and high computational costs.

On another route, Son et al. [42] introduced a new form



(a) (Traditional) Mesh (b) DMesh, DMesh++

Figure 2. **Conceptual comparison of traditional mesh and variants of DMesh [42].** Traditional meshes employ a non-differentiable, discrete data structure, $\mathbb{F}$, to store vertex indices that define connectivity, whereas DMesh++ encodes connectivity and additional information into continuous point-wise features, $\mathbb{P}$. Mesh generated from DMesh++ avoids several degeneracies—such as self-intersections and thin triangles—that can compromise its suitability for downstream applications.

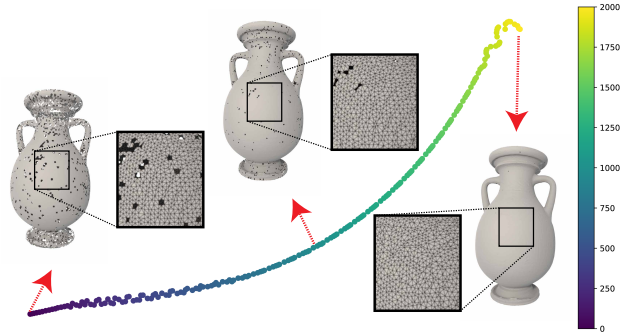


Figure 3. **Mesh topology change during 3D point cloud reconstruction of a vase.** We visualize the evolution of point-wise features by projecting them onto a 2-dimensional space using T-SNE [43] at each optimization step (ranging from 0 to 2K). As these features continuously evolve, the mesh undergoes discrete topological changes to progressively recover the target shape.

of differentiable mesh called DMesh, which is essentially a probabilistic approach. For a given set of points, they explicitly compute probabilities for possible face combinations to exist on the mesh based on the continuous point-wise features. This approach minimizes several mesh degeneracies, and is free from outliers, as it is not data-driven (Fig. 2). Therefore, this probabilistic approach opens up a new venue to adopt meshes in a machine learning pipeline, such as generative models [47, 51]. However, it suffers from excessive computational cost when the number of points increases (Fig. 7), which limits its applicability for representing complex shapes with detailed structures.

In this work, we introduce DMesh++, which overcomes the computational limitations of DMesh while retaining its core advantages. To that end, we present *Minimum-Ball* algorithm. While the computational cost to evaluate face probability is $O(N)$ for DMesh, where N is the number of points that define the mesh, our *Minimum-Ball* algorithm has $O(\log N)$ computational cost (Sec. 3.2 and Fig. 7).

The direct application of DMesh++ is a reconstruction task. It effectively reconstructs complex 2D and 3D meshes of diverse topology from point clouds or multi-view images (Figs. 1, 9 and 10). During the optimization of continuous point-wise features, we observe dynamic topological changes in the mesh that recover the target shape (Fig. 3).

To summarize, our contributions are the following:

- We present DMesh++, an enhanced version of DMesh [42], which overcomes its computational bottlenecks by employing the *Minimum-Ball* algorithm.
- We propose a reconstruction algorithm that incorporates efficient loss formulations and additional mesh operations to effectively recover 2D and 3D shapes from point clouds or multi-view images.
- We validate our approach on 500+ mesh models with diverse topology, which are collected from Thingi10K [52] and Objaverse [10] dataset.

2. Related Work

While meshes offer an efficient and flexible representation of shapes, they are mainly constrained by their connectivity issues, which limit their applicability in machine learning. To address these challenges, shape inference in machine learning has evolved through three stages.

Using Alternative Differentiable Shape Representations. Rather than handling mesh directly, some prior work extract mesh from alternative differentiable shape representations. Neural implicit representations, like (un)signed distance fields [22, 24, 32, 34, 44–46, 49, 50], encode distance fields in neural networks, and use iso-surface extraction algorithms [12, 16, 25] to generate the final mesh. Another method encodes distance directly into spatial points and applies differentiable iso-surface extraction [20, 23, 28, 29, 38, 39, 46]. While often more efficient, these methods typically cannot handle open surfaces; though [23] does, it cannot represent non-orientable geometries. Gaussian Splatting [18] also encodes visual data as spatial "splats" but lacks the geometric accuracy of implicit functions.

Inferring Meshes Differentiably. The main challenge in differentiable mesh handling is the exponential growth of possible vertex connections as vertex count increases. To simplify this challenge, most prior works assume fixed registration and permit only local connectivity changes [4, 19, 21, 31, 33, 53]. Recently, data-driven approaches have aimed to overcome these limitations by training generative models [5, 6, 40, 41] that predict vertex connectivity from point clouds. Specifically, SpaceMesh [40] ensures combinatorial manifold mesh generation. However, these models struggle with outliers and self-intersections.

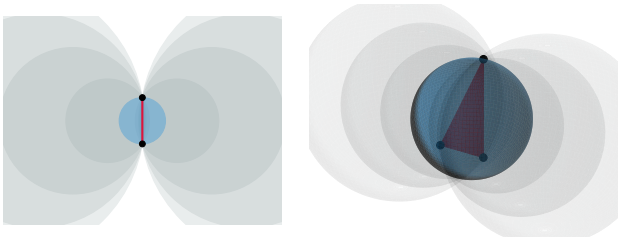


Figure 4. **Bounding balls of a face F (red) in 2D (left) and 3D (right).** The minimum bounding ball (B_F) is rendered in blue, while the others are rendered in gray.

Designing A Differentiable Form of Mesh. Son et al. [42] recently introduced DMesh, a differentiable mesh formulation using a probabilistic approach. DMesh augments each point with two continuous values, along with its position (Fig. 2), and applies a "tessellation" function in Eq. (1) to deterministically generate a mesh from a point set. This method adapts to various geometric topologies, including non-orientable open surfaces, and avoids self-intersections. With DMesh, optimizing or inferring only point-wise features is sufficient to generate the mesh.

However, DMesh's tessellation function is slow due to its reliance on Weighted Delaunay Triangulation (WDT), which has a practical time complexity of $O(N)$ for N points using the CGAL package [14]. For $N = 100K$ in 3D, the runtime can reach up to 800 milliseconds (Fig. 7), limiting DMesh's applicability for complex shapes requiring finer detail. Moreover, it is hardly possible to accelerate WDT using parallelization, because of its inherent race conditions. Therefore, in this work, we eliminate WDT, and propose a more efficient differentiable mesh formulation.

3. Formulation

In this section, we first provide the high-level formulation for computing probability of a face to exist in the mesh. Then, we introduce *Minimum-Ball* (Sec. 3.2), which is our primary algorithm.

3.1. Preliminary

In this work, we refer to a $(d-1)$-simplex in d-dimensional space as a "face" (e.g., a line segment for $d = 2$ or a triangle for $d = 3$). DMesh [42] tessellates d-dimensional convex space using faces, with the actual surface on "real" faces and "imaginary" faces enclosing the "real" part to support the convex space (Fig. 1).

In DMesh, each point is a $(d + 2)$-dimensional vector: the first d values denote position, while the remaining two represent the Weighted Delaunay Triangulation (WDT) weight (w) [1] and real value (ψ). The $\psi \in [0, 1]$ of a point indicates whether it lies on the shape, specifically if $\Psi(p) > 0.5$, where $\Psi(p)$ is ψ of a point p.

For a point set $\mathbb{P}$, let $\mathbb{F}_{wdt}$ represent the faces in WDT of $\mathbb{P}$. DMesh then introduces a **"tessellation"** function to determine if a face F exists on the mesh:

$$T_{DMesh}(\mathbb{P}, F) = (F \in \mathbb{F}_{wdt}) \wedge (\min_{p \in F} \Psi(p) > 0.5). \quad (1)$$

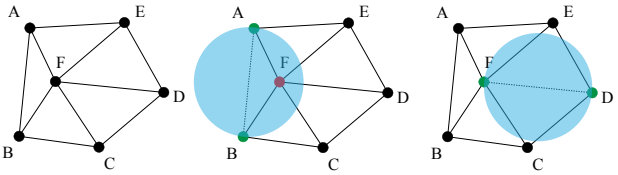


Figure 5. **Minimum-Ball condition in 2D.** In the left, 2D Delaunay Triangulation (DT) of 6 points is given. In middle and right figure, we render B_F for two faces ($\overline{AB}$, $\overline{DF}$) in blue.

DMesh++ introduces an alternative tessellation function for faster processing. By removing the need for WDT, we eliminate the WDT weight and represent each point as a $(d + 1)$-dimensional vector, $(x_1, ..., x_d, \psi)$[1]. In place of WDT, we implement a faster scheme called the *Minimum-Ball* condition (Definition 3.1) for defining the tessellation function. Letting $\mathbb{F}_{min}$ represent the set of faces that meet this condition, we define the tessellation function as

$$T_{DMesh++}(\mathbb{P}, F) = (F \in \mathbb{F}_{min}) \wedge (\min_{p \in F} \Psi(p) > 0.5). \quad (2)$$

In our differentiable framework, we compute probability of F to satisfy these two conditions: Λ_{min} and Λ_{real}, respectively. Then, we compute the final probability of F to exist on the mesh as $\Lambda(F) = \Lambda_{min}(F) \times \Lambda_{real}(F)$. For Λ_{real}, we use differentiable $\min$ operator as DMesh. In the next section, we explain how we define Λ_{min}.

3.2. Minimum-Ball Algorithm

The mesh generated by DMesh's tessellation function in Eq. (1) is free from self-intersections because $\mathbb{F}_{wdt}$ itself is free from them. Additionally, it minimizes the occurrence of thin triangles—undesirable in many downstream tasks such as physics simulations—thanks to the properties of the WDT. However, computing the tessellation function is computationally expensive, as it requires calculating the WDT to define $\mathbb{F}_{wdt}$. In designing our *Minimum-Ball*, we aimed to eliminate this computational bottleneck while preserving the favorable properties related to self-intersection avoidance and triangle quality. In the following, we demonstrate that *Minimum-Ball* satisfies these requirements.

For a given set of points $\mathbb{P} \in \mathbb{R}^d$ and a face $F = \{p_1, p_2, ..., p_d\} \subset \mathbb{P}$, we define a bounding ball of F as a d-dimensional ball that goes through every point of F. Note that this bounding ball is not unique, but there exists a unique minimum bounding ball, which has the minimum radius among every bounding ball. We name it as B_F (Fig. 4). Then, we define $\mathbb{F}_{min}$ as a set of faces whose minimum bounding ball does not contain any other point in $\mathbb{P}$.

[1]Points could carry additional features, such as color (Fig. 2).

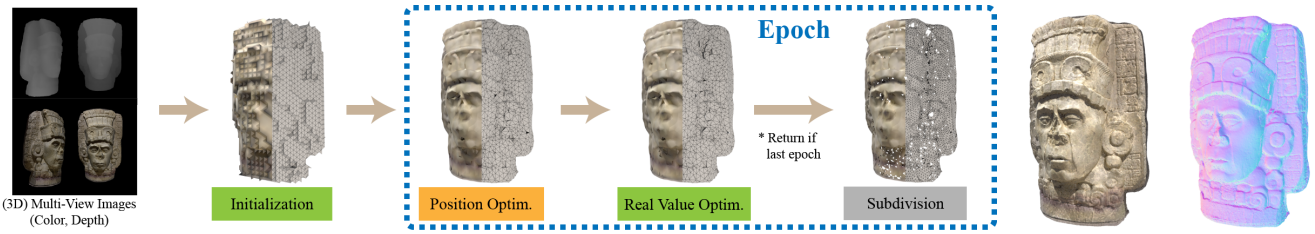


Figure 6. **Reconstruction process for 3D multi-view colored images of a sculpture.** In each stage, we optimize different per-point features: the position and the real (ψ), while the per-point color is refined at every stage. (Left) We display the meshes at each stage during the first epoch. (Right) We provide a rendering of the final mesh along with its view-point normal, which reveals the fine mesh details.

Definition 3.1. $F \in \mathbb{F}_{min}$ if and only if there is no point in $\mathbb{P}$ that lies (strictly) inside B_F.

Note that we can ignore points in F, as they are located on the boundary of B_F. In Fig. 5, we render a 2D case, where $\overline{AB}$ does not satisfy this definition because of F. In contrast, $\overline{DF}$ satisfies this condition. Then, we can observe that $\mathbb{F}_{min}$ is a subset of faces in Delaunay Triangulation (DT) of $\mathbb{P}$ ($\mathbb{F}_{dt}$).

Lemma 3.2. $F \in \mathbb{F}_{min} \Rightarrow F \in \mathbb{F}_{dt}$.

Proof. By definition, a face F is in $\mathbb{F}_{dt}$ if there is a bounding ball of F that does not contain any other point in $\mathbb{P}$ [8]. If the face F is in $\mathbb{F}_{min}$, its minimum bounding ball satisfies this condition. Thus F is in $\mathbb{F}_{dt}$. □

Note that $\mathbb{F}_{dt}$ is also free from self-intersections as $\mathbb{F}_{wdt}$, and thus is $\mathbb{F}_{min}$. Furthermore, it inherently minimizes the number of thin triangles, as guaranteed by DT. However, note that $\mathbb{F}_{min}$ does not necessarily tessellate the entire convex shape, as there could be faces in $\mathbb{F}_{dt}$ that are not in $\mathbb{F}_{min}$ (*e.g.* $\overline{AB}$ in Fig. 5).

Now, based on Definition 3.1, we can check if F is in $\mathbb{F}_{min}$. Let us denote the center and radius of B_F as $B_F^c \in \mathbb{R}^d$ and B_F^r. We can compute these values in a differentiable way (Appendix 7.2). Then, we can compute the signed distance between B_F and $\mathbb{P}$ as follows:

$$d(B_F, \mathbb{P}) = \min_{p \in \mathbb{P}-F} ||p - B_F^c|| - B_F^r. \tag{3}$$

As shown above, we can easily find $d(B_F, \mathbb{P})$ by finding the nearest point of B_F^c in $\mathbb{P}-F$. Using this signed distance, we can check if F is in $\mathbb{F}_{min}$ as follows.

$$F \in \mathbb{F}_{min} \Leftrightarrow d(B_F, \mathbb{P}) > 0. \tag{4}$$

Then, we define Λ_{min} with sigmoid function as

$$\Lambda_{min}(F) = \sigma(d(B_F, \mathbb{P}) \cdot \alpha_{min}), \tag{5}$$

where α_{min} is a constant (Appendix 7.3).

With this formulation, we can evaluate Eq. (2) far more efficiently than Eq. (1). Our method relies on a highly parallelizable nearest neighbor search algorithm[2], unlike the sequential WDT. While WDT has a practical time complexity of $O(|\mathbb{P}|)$, it is relatively slow. In contrast, our approach has a time complexity of $O(|F| \cdot \log |\mathbb{P}|)$, where $|F|$ is the number of query faces to evaluate. However, by parallelizing the nearest neighbor search across query faces, especially on GPU, this complexity effectively reduces to $O(\log |\mathbb{P}|)$[3]. This allows our tessellation function to run up to 32 times faster in 3D than DMesh [42] (Fig. 7). For optimization tasks like reconstruction, we further accelerate by periodically caching nearest neighbors for each query face (Appendix 7.4). We provide formal algorithm in Appendix 7.1.

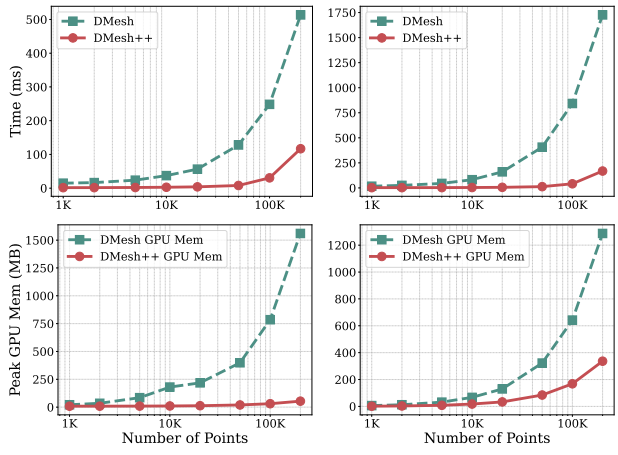
4. Reconstruction Process

The goal of reconstruction is to optimize point-wise features so that the resulting mesh aligns with the input observation. As shown in Fig. 3, the discrete mesh topology dynamically changes during the optimization of continuous features to better fit the given input.

Fig. 6 provides an overview of our reconstruction process for recovering a 3D colored mesh from multi-view images. In the first stage, we initialize per-point features. If a point cloud is available, we use it to obtain a better starting point (see the initial mesh in Fig. 3); otherwise, we initialize with a regular tetrahedral grid. Next, we optimize the point positions while keeping the real values fixed, followed by optimizing the real values while fixing the point positions. In these two stages, we minimize loss functions tailored to each input modality (e.g., Chamfer Distance loss for point clouds, rendering loss for images). To increase mesh complexity and capture finer details, we subdivide the mesh by inserting additional points. This process is iterated for a fixed number of epochs. Finally, to accelerate the overall process and enhance the final mesh quality, we introduce several innovations at each stage. Detailed explanations are provided in Appendix 8, due to lack of space.

[2]We used implementation of PyTorch3D [37].

[3]We assume that $|F|$ does not increase exponentially, which is a practical assumption as query faces are often determined by local proximity.



(a) 2D (b) 3D

Figure 7. **Comparison of tessellation cost.** Our method computes face probabilities up to 16 times faster in 2D and 32 times faster in 3D than DMesh [42], while using up to 96% less GPU memory in 2D and 75% less in 3D.

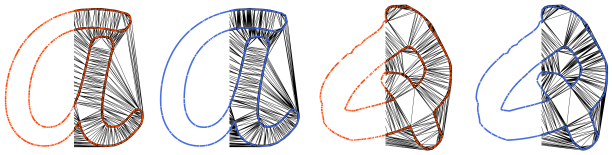
5. Experiments

This section presents our experimental results. First, we evaluate how *Minimum-Ball* enhances the computational efficiency of the tessellation function in both 2D and 3D. Next, we demonstrate the results of a point cloud reconstruction task in 2D and 3D as a practical application. Finally, we showcase an application in 3D multi-view reconstruction. Together, these results illustrate that our method is well-suited for downstream tasks involving complex shapes. Our main algorithms are implemented in PyTorch [35] and CUDA [30]. All experiments were conducted on a system with an AMD EPYC 7R32 CPU and an NVIDIA A10 GPU. For details, refer to Appendix 9.

5.1. Tessellation Cost

We compare the computational cost of the tessellation function for DMesh [42] (Eq. (1)) and our DMesh++ (Eq. (2)). For both 2D and 3D scenarios, we randomly generate N points within a unit cube, find each point's 10 nearest neighbors, and use these proximities to form potential face combinations. From these, we randomly select N faces as query faces for the tessellation function. For each value of N (ranging from $1K$ to $200K$ to reflect practical scenarios), we conducted 5 trials and averaged the computational costs.

In Fig. 7, we compare the computational costs of DMesh and DMesh++. In terms of speed, DMesh's performance scales linearly with the number of points in both 2D and 3D due to its sequential WDT algorithm. In contrast, DMesh++ exhibits sub-linear scaling up to 50K points, benefiting from GPU parallelization (Sec. 3.2). Beyond this range, computational costs increase more sharply because of GPU thread limitations; however, DMesh++ still processed 200K points in $117ms$ for 2D and $168ms$ for 3D. Regarding GPU memory usage, both methods scale linearly, but DMesh++ uses significantly less memory since it stores only the additional information related to the minimum balls, whereas DMesh must store all details of the power diagram on the GPU[4].



(a) Pacifico, "a" (b) Permanent Marker, "Q"

Figure 8. **Qualitative comparison of 2D point cloud reconstruction results.** The outputs of DMesh [42] and DMesh++ are rendered in red and blue, respectively.

Method	CD($\times 10^{-6}$)↓	# Verts.	# Edges.	Time (sec)
DMesh [42]	1.97	2506	2245	30.39
DMesh++	1.82	2862	2793	11.33

Table 1. **Quantitative comparison of the 2D point cloud reconstruction task for the font dataset.** DMesh++ reconstructs 2D meshes with greater accuracy and efficiency than DMesh.

These results demonstrate that the *Minimum-Ball* algorithm significantly enhances tessellation efficiency, enabling the effective handling of complex shapes.

5.2. Reconstruction Tasks

Dataset. For the 2D point cloud reconstruction task, we used vector graphics of 26 letters from four different font styles downloaded from the Google Fonts service[5]. Additionally, we employed six vector graphic images representing complex drawings from Adobe Stock[6].

For the 3D reconstruction tasks, we randomly selected 500 models from Thingi10K [52] dataset, half of which are closed-surface models and the other half are open-surface models. Also, we manually chose 20 and 30 models from Thingi10K and Objaverse [10] dataset for better diversity. We normalized the model scale to 1 before reconstruction.

Metrics. For 2D results, we report the Chamfer Distance (CD) of the reconstructed outputs for quantitative comparison. For 3D, we assess geometric accuracy using five metrics—CD (Chamfer Distance), F1 (F1 score), NC (Normal Consistency), ECD (Edge Chamfer Distance), and EF1 (Edge F1 score)—following [7]. Additionally, we evaluate mesh quality using four metrics: AR (Aspect Ratio), SI (Self-Intersection ratio), NME (Non-Manifold Edge ratio), and NMV (Non-Manifold Vertex ratio). Finally, we provide statistics on mesh complexity (e.g., vertex and face counts) along with the reconstruction time.

[4]Note that the 2D version of DMesh requires more memory than the 3D version, as the 2D implementation is not as optimized using CUDA.

[5]https://fonts.google.com/

[6]https://stock.adobe.com/

	Geometric Accuacy					Mesh Quality				Statistics		
Method	CD($\times 10^{-3}$)↓	F1↑	NC↑	ECD↓	EF1↑	AR↓	SI↓	NME↓	NMV↓	# Verts.	# Faces.	Time (sec)
VoroMesh [26]	19.591	0.352	0.855	0.054	0.072	145.7	0	0.001	0	64561	129338	11
DMesh++	0.034	0.471	0.919	0.063	0.094	1.765	0	0.130	0.003	25415	58537	282
PSR [17]	10.164	0.392	0.943	0.302	0.026	5.218	0	0	0	139857	279739	4
PoNQ [27]	1.578	0.402	0.934	0.056	0.090	2.288	0	0.002	0	47254	94664	32
DMesh [42]	0.154	0.289	0.921	0.077	0.069	1.961	0	0.103	0.002	5815	13088	1147
DMesh++	0.033	0.480	0.938	0.060	0.116	1.814	0	0.087	0.004	25396	55546	282

Table 2. **Quantitative comparison of 3D point cloud reconstruction results over 50 manually chosen models from Thingi10K [52] and Objaverse [10].** We highlight the best results for each metric. The upper panel compares methods that use *unoriented* point clouds (VoroMesh, DMesh++), while the lower panel displays methods that use *oriented* point clouds (PSR, PoNQ, DMesh, DMesh++).

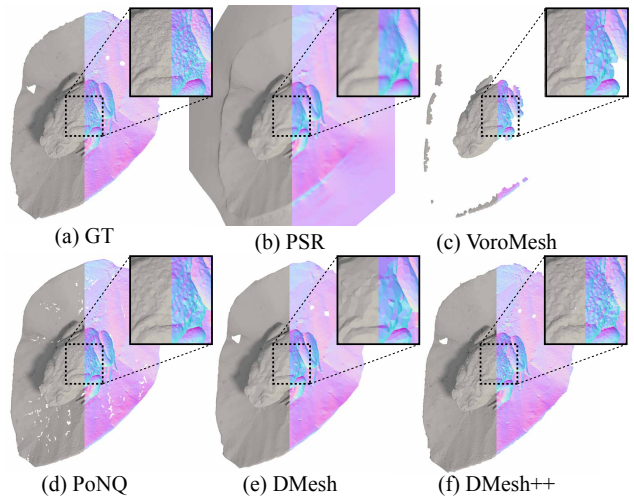


Figure 9. **Qualitative comparison of 3D point cloud reconstruction results for a toad sitting on a leaf.** For each result, we render its diffuse image on the left, and view-point normals on the right.

Method	CD($\times 10^{-4}$)↓	F1↑	NC↑	ECD↓	EF1↑
PSR [17]	12.1 / 16.3	0.47 / 0.45	0.97 / 0.96	0.45 / 0.37	0.01 / 0.01
PoNQ [27]	2.44 / 7.86	0.48 / 0.45	0.97 / 0.95	0.01 / 0.04	0.26 / 0.22
DMesh [42]	2.82 / 3.29	0.25 / 0.22	0.94 / 0.93	0.01 / 0.01	0.15 / 0.12
DMesh++	0.37 / 0.37	0.48 / 0.47	0.96 / 0.95	0.02 / 0.02	0.25 / 0.23

Table 3. **Quantitative comparison of 3D point cloud reconstruction results over 500 randomly chosen models from Thingi10K [52].** The results over closed surfaces and open surfaces are shown together: (closed / open). We highlight the best results for two different categories with red and blue, respectively.

5.2.1. 2D Point Cloud Reconstruction

In this task, we aim to reconstruct a 2D mesh from a 2D point cloud to demonstrate that the DMesh++ formulation is applicable in 2D. We primarily compare our reconstruction results on a font dataset with those obtained by DMesh [42], whose formulation is also easily extendable to 2D. As input, we sampled 1,000 points from each spline curve composing the font and downsampled the entire point cloud using a grid with a cell size of 0.005.

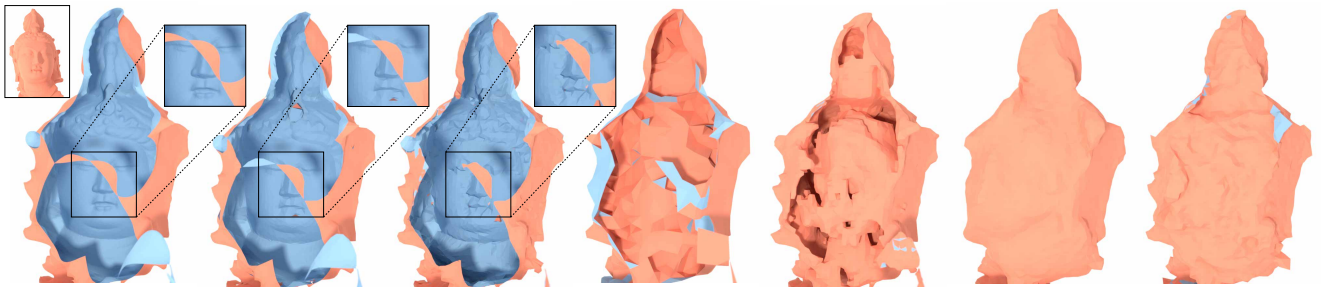
In Tab. 1, we present a quantitative comparison of the reconstruction results. We observe that DMesh++ reconstructs 2D meshes more faithfully than DMesh in terms of CD loss, while also running 2.6× faster. In Fig. 8, we render the reconstructed 2D meshes for two examples. The results show that DMesh++ produces significantly fewer holes compared to DMesh, which is consistent with the CD loss comparison. However, some of these holes are inevitable, as we mainly reconstruct our shape with CD loss and there are places that lack points.

Additionally, we reconstructed several complex 2D drawings from point clouds to demonstrate the computational efficiency of DMesh++. Since these drawings contain finer details than the fonts, we downsampled the entire point cloud using a grid with a cell size of 0.001. When we attempted to reconstruct these drawings with DMesh, the GPU memory consumption became prohibitively high, resulting in an error during reconstruction. Therefore, for qualitative evaluation, we report only the results of DMesh++ in Fig. 1 and Appendix 9.2.2.

Before moving on, we'd like to introduce an experimental algorithm that is applicable to 2D mesh optimization called the *Reinforce-Ball* algorithm, which could be used for producing an efficient mesh that adapts to local geometry. Please see Appendix 10 for more details.

5.2.2. 3D Point Cloud Reconstruction

In this task, we reconstruct a 3D mesh from a dense 3D point cloud. As input, we sampled 200K points from the ground truth mesh using the Poisson disk sampling algorithm [3] implemented in MeshLab [9]. For comparison, we employed other widely used optimization-based point cloud reconstruction methods, including Screened Poisson Surface Reconstruction (PSR)[17], VoroMesh[26], PoNQ [27], and DMesh [42]. For PSR, DMesh, and PoNQ, we used oriented point clouds for reconstruction, while for DMesh++ we tested both unoriented and oriented point clouds. We used the PSR implementation available in MeshLab. For DMesh, we used its default settings; for VoroMesh and PoNQ, we employed a grid size of 128 and optimized for 1000 epochs to achieve the best results.



(a) Ground Truth (b) DMesh++ (c) GShell (d) DMesh (e) FlexiCubes (f) DMTet (g) Remeshing

Figure 10. **Qualitative comparison of 3D multi-view reconstruction results for open surface.** Here we illustrate from back of an open surface model (the front view is rendered at the left top of (a)). Colors represent inside and outside facing surfaces.

	Geometric Accuacy					Mesh Quality				Statistics		
Method	CD($\times 10^{-3}$)↓	F1↑	NC↑	ECD↓	EF1↑	AR↓	SI↓	NME↓	NMV↓	# Verts.	# Faces.	Time (sec)
Remeshing [33]	0.977	0.343	0.907	0.088	0.044	1.562	0.220	0	0	13273	26540	25
DMTet [38]	1.395	0.191	0.868	0.145	0.032	6.175	0	0	0	20549	41131	201
FlexiCubes [39]	3.493	0.290	0.880	0.091	0.038	2.093	0.011	0	0.005	14811	28882	88
GShell [23]	5.807	0.326	0.904	0.116	0.047	2.793	0.047	0	0.004	14587	28054	209
DMesh [42]	0.697	0.328	0.898	0.104	0.045	1.820	0	0.042	0.002	2461	5058	772
DMesh++	0.342	0.360	0.923	0.074	0.059	1.639	0	0.025	0.036	12507	23727	205

Table 4. **Quantitative comparison of 3D multi-view reconstruction results over 50 manually chosen models from Thingi10K [52] and Objaverse [10].** We highlight the best results and the second best results for each evaluation metric.

We report the quantitative comparisons in Tabs. 2 and 3. In Tab. 2, results are averaged over the 50 manually chosen models and split into two categories: methods using *un-oriented* versus *oriented* point clouds. In both categories, DMesh++ performs best or is comparable to other methods in geometric accuracy and triangle aspect ratio. Compared to DMesh, DMesh++ outperforms all metrics while handling 4.2× more faces with a 76% reduction in computation time. Furthermore, DMesh++ achieves significantly better CD, F1, and EF1 scores than PSR, VoroMesh, and PoNQ, which struggle with open surfaces. Separate evaluations in Tab. 3 over 500 randomly chosen models confirm that DMesh++ largely outperforms on open surfaces and performs comparably or better on closed surfaces.

These quantitative results align with the qualitative comparison in Fig. 9, where a toad sitting on a thin leaf is reconstructed . While PSR and VoroMesh fail to reconstruct the open surface of the leaf and PoNQ produces many holes, DMesh captures the overall geometry but misses fine details. DMesh++ successfully recovers both the overall shape and the intricate details. For qualitative comparison on a closed surface, see Fig. 19.

5.2.3. 3D Multi-View Reconstruction

In this task, we reconstruct a mesh from multi-view images of a target object, assuming full knowledge of the rendering model and lighting conditions. Specifically, we employ simple Phong shading [36] with a directional light from the camera to the object for rendering the ground truth images.

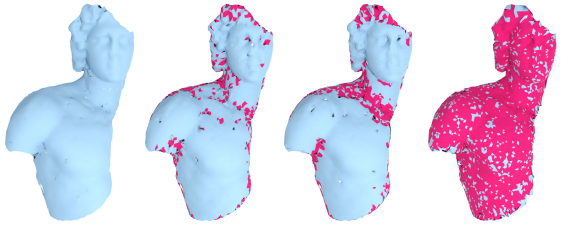
Method	CD($\times 10^{-4}$)↓	F1↑	NC↑	ECD↓	EF1↑
REM [33]	58.0 / 27.5	0.33 / 0.32	0.90 / 0.90	0.06 / 0.07	0.11 / 0.09
DMT [38]	36.2 / 65.1	0.21 / 0.21	0.89 / 0.88	0.17 / 0.13	0.03 / 0.03
FLE [39]	14.5 / 34.1	0.37 / 0.35	0.92 / 0.91	0.04 / 0.06	0.10 / 0.08
GSH [23]	5.74 / 54.6	0.38 / 0.36	0.94 / 0.93	0.04 / 0.06	0.15 / 0.12
DME [42]	11.9 / 11.7	0.18 / 0.20	0.87 / 0.87	0.06 / 0.08	0.04 / 0.04
DMesh++	3.35 / 3.82	0.37 / 0.36	0.94 / 0.93	0.03 / 0.03	0.21 / 0.17

Table 5. **Quantitative comparison of 3D multi-view reconstruction results over 500 randomly chosen models from Thingi10K [52].** The results over closed surfaces and open surfaces are shown together: (closed / open). We highlight the best results for two different categories with red and blue, respectively.

We generate (512×512) diffuse and depth maps of the object from 64 viewpoints (Fig. 20) to supervise the reconstruction process. Note that we omit colors and textures in this experiment to focus solely on geometric quality.

For comparison, we evaluated five mesh reconstruction algorithms: Remeshing [2], DMTet [38], FlexiCubes [39], GShell [23], and DMesh [42]. We optimized each method to produce the best-quality meshes with similar vertex and face counts. Specifically, we set the grid sizes to 128 for DMTet, and 80 for both FlexiCubes and GShell. For more details, please refer to Appendix 9.4.

In Tabs. 4 and 5 and Figs. 10 and 11, we present quantitative and qualitative comparisons of the reconstruction results. In Tab. 4, we observe that Remeshing, DMTet, and FlexiCubes have high CD errors, largely because they cannot represent open surfaces (Fig. 10). This limitation also explains why Remeshing and FlexiCubes are faster than



(a) DMesh++ (b) GShell (c) FlexiCubes (d) Remeshing

Figure 11. **Self-intersection of the reconstructed mesh.** The self-intersected faces of the mesh are rendered in red.

other methods. Although Remeshing achieved the best AR and avoided non-manifoldness, it generated numerous self-intersections, particularly on open surfaces (Fig. 11).

When it comes to GShell, we found out that it usually improves CD loss by representing open surfaces through sub-surface extraction from closed templates. However, it still struggles with complex open surfaces (Fig. 10). Also, it had a robustness issue with several outlier models – which is the main reason of high average CD, even though it achieved much better CD for most of the models. Additionally, it also suffers from self-intersections (Fig. 11) and suboptimal AR. Compared to that, DMesh produced self-intersection free mesh with better CD and AR, as it can represent open surfaces robustly. Also, it produced much simpler mesh than the other methods. However, it sometimes produced false inner structure due to occlusion (Fig. 10), and its largest drawback was in computational cost, limiting its utility for fine-grained reconstructions.

DMesh++ addresses this issue using the *Minimum-Ball* algorithm (Sec. 3.2) and nearest neighbor caching (Appendix 7.4). It achieves the best or comparable results across all metrics while maintaining computational costs similar to those of other methods. Furthermore, as shown in Tab. 5, DMesh++ delivers superior, or at least comparable performance for both closed and open surfaces. Qualitative comparisons in Figs. 10 and 20 also support this observation. These results prove the robustness of DMesh++ in handling complex shapes with diverse topologies, whereas other methods are limited in one aspect or another.

Colors. As mentioned in Fig. 2 and Sec. 3.1, points can carry additional features. Here, we demonstrate that we can jointly optimize per-point colors to recover a textured shape from multi-view images. For a point on a face, the color of the point is determined by barycentric interpolation of the face vertex colors. Under the assumption that we know all the rendering and lighting conditions, we can reconstruct textured meshes from multi-view images as shown in Figs. 1, 6 and 12. In particular, Fig. 12 demonstrates that DMesh++ can reconstruct a small scene on which physics simulations, such as bouncing balls, can be run directly. These results highlight a promising future direction of jointly optimizing additional per-point features.

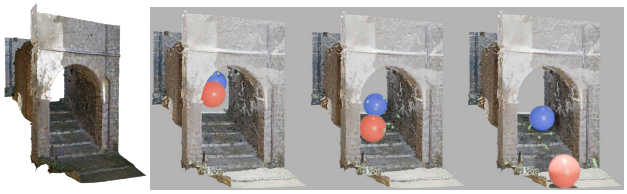


Figure 12. **Physics simulation on a staircase reconstructed from multi-view images.** We simulate the motion of bouncing balls directly on the mesh generated by DMesh++.

6. Conclusion

We presented DMesh++, a probabilistic approach for efficient, differentiable mesh connectivity handling. Our *Minimum-Ball* algorithm significantly reduces computational costs, enabling DMesh++ to recover 2D and 3D shapes with diverse topologies from point clouds or multi-view images more effectively than baseline methods.

Limitations. There are areas where DMesh++ can be further improved. First, some non-manifoldness errors remain in the reconstruction results (Tabs. 2 and 4), while observing significant improvement over DMesh [42]. We conjecture that there is a trade-off between the representation power and the occurrence of these topological errors; more careful analysis of the trade-off is needed to maintain expressiveness while eliminating errors. Second, there are application-specific challenges. Although our method yields superior 3D point cloud reconstruction results, it incurs higher computational costs (Tab. 2), and it is less effective for sparse point clouds. In 3D multi-view reconstruction, we cannot use the current implementation for real-world images, as discussed in Appendix 9.4.2.

Future Directions. For 3D point cloud reconstruction, we plan to explore alternative loss formulations to CD loss, as its computation currently dominates the overall cost.

To extend our 3D multi-view reconstruction algorithm to real-world images, we will integrate DMesh++ with alternative representations such as Gaussian Splatting (GS) [13, 18]. By appending GS features to the per-point features and optimizing them jointly, we expect to recover high-quality meshes that we can readily use for downstream tasks.

Furthermore, we envision leveraging DMesh++ to train generative models that capture complex mesh connectivity. Currently, the *Minimum-Ball* algorithm only uses point positions for connectivity. In future work, we plan to incorporate additional per-point features into this algorithm to train generative models capable of understanding diverse and intricate mesh connectivity, such as that of DNA (Fig. 1).

We believe our work establishes an important foundation for harnessing differentiable, probabilistic mesh within the current optimization framework, and we hope it will further drive future downstream applications.

References

[1] Franz Aurenhammer. Power diagrams: properties, algorithms and applications. *SIAM Journal on Computing*, 16 (1):78–96, 1987. 3

[2] Jonathan W Brandt and V Ralph Algazi. Continuous skeleton computation by voronoi diagram. *CVGIP: Image understanding*, 55(3):329–338, 1992. 7

[3] Robert Bridson. Fast poisson disk sampling in arbitrary dimensions. *SIGGRAPH sketches*, 10(1):1, 2007. 6

[4] Wenzheng Chen, Huan Ling, Jun Gao, Edward Smith, Jaakko Lehtinen, Alec Jacobson, and Sanja Fidler. Learning to predict 3d objects with an interpolation-based differentiable renderer. *Advances in neural information processing systems*, 32, 2019. 2

[5] Yiwen Chen, Tong He, Di Huang, Weicai Ye, Sijin Chen, Jiaxiang Tang, Xin Chen, Zhongang Cai, Lei Yang, Gang Yu, et al. Meshanything: Artist-created mesh generation with autoregressive transformers. *arXiv preprint arXiv:2406.10163*, 2024. 1, 2

[6] Yiwen Chen, Yikai Wang, Yihao Luo, Zhengyi Wang, Zilong Chen, Jun Zhu, Chi Zhang, and Guosheng Lin. Meshanything v2: Artist-created mesh generation with adjacent mesh tokenization. *arXiv preprint arXiv:2408.02555*, 2024. 1, 2

[7] Zhiqin Chen, Andrea Tagliasacchi, Thomas Funkhouser, and Hao Zhang. Neural dual contouring. *ACM Transactions on Graphics (TOG)*, 41(4):1–13, 2022. 5

[8] Siu-Wing Cheng, Tamal Krishna Dey, Jonathan Shewchuk, and Sartaj Sahni. *Delaunay mesh generation*. CRC Press Boca Raton, 2013. 4

[9] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, Guido Ranzuglia, et al. Meshlab: an open-source mesh processing tool. In *Eurographics Italian chapter conference*, pages 129–136. Salerno, Italy, 2008. 6

[10] Matt Deitke, Dustin Schwenk, Jordi Salvador, Luca Weihs, Oscar Michel, Eli VanderBilt, Ludwig Schmidt, Kiana Ehsani, Aniruddha Kembhavi, and Ali Farhadi. Objaverse: A universe of annotated 3d objects. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13142–13153, 2023. 2, 5, 6, 7

[11] Evan Greensmith, Peter L Bartlett, and Jonathan Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5(9), 2004. 10

[12] Benoit Guillard, Federico Stella, and Pascal Fua. Meshudf: Fast and differentiable meshing of unsigned distance field networks. In *European Conference on Computer Vision*, pages 576–592. Springer, 2022. 2

[13] Binbin Huang, Zehao Yu, Anpei Chen, Andreas Geiger, and Shenghua Gao. 2d gaussian splatting for geometrically accurate radiance fields. In *ACM SIGGRAPH 2024 conference papers*, pages 1–11, 2024. 8

[14] Clément Jamin, Sylvain Pion, and Monique Teillaud. 3D triangulations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023. 3

[15] Rasmus Jensen, Anders Dahl, George Vogiatzis, Engin Tola, and Henrik Aanæs. Large scale multi-view stereopsis evaluation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 406–413, 2014. 7, 8

[16] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of hermite data. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 339–346, 2002. 2

[17] Michael Kazhdan and Hugues Hoppe. Screened poisson surface reconstruction. *ACM Transactions on Graphics (ToG)*, 32(3):1–13, 2013. 6

[18] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42 (4), 2023. 2, 8

[19] Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. Modular primitives for high-performance differentiable rendering. *ACM Transactions on Graphics (TOG)*, 39(6):1–14, 2020. 2, 3

[20] Yiyi Liao, Simon Donne, and Andreas Geiger. Deep marching cubes: Learning explicit surface representations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2916–2925, 2018. 2

[21] Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. Soft rasterizer: A differentiable renderer for image-based 3d reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7708–7717, 2019. 2

[22] Yu-Tao Liu, Li Wang, Jie Yang, Weikai Chen, Xiaoxu Meng, Bo Yang, and Lin Gao. Neudf: Leaning neural unsigned distance fields with volume rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 237–247, 2023. 2

[23] Zhen Liu, Yao Feng, Yuliang Xiu, Weiyang Liu, Liam Paull, Michael J Black, and Bernhard Schölkopf. Ghost on the shell: An expressive representation of general 3d shapes. *arXiv preprint arXiv:2310.15168*, 2023. 2, 7, 6

[24] Xiaoxiao Long, Cheng Lin, Lingjie Liu, Yuan Liu, Peng Wang, Christian Theobalt, Taku Komura, and Wenping Wang. Neuraludf: Learning unsigned distance fields for multi-view reconstruction of surfaces with arbitrary topologies. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 20834–20843, 2023. 2

[25] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Seminal graphics: pioneering efforts that shaped the field*, pages 347–353. 1998. 2

[26] Nissim Maruani, Roman Klokov, Maks Ovsjanikov, Pierre Alliez, and Mathieu Desbrun. Voromesh: Learning watertight surface meshes with voronoi diagrams. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 14565–14574, 2023. 6

[27] Nissim Maruani, Maks Ovsjanikov, Pierre Alliez, and Mathieu Desbrun. Ponq: a neural qem-based mesh representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3647–3657, 2024. 6, 7

[28] Ishit Mehta, Manmohan Chandraker, and Ravi Ramamoorthi. A level set theory for neural implicit evolution under

explicit flows. In *European Conference on Computer Vision*, pages 711–729. Springer, 2022. 2

[29] Jacob Munkberg, Jon Hasselgren, Tianchang Shen, Jun Gao, Wenzheng Chen, Alex Evans, Thomas Müller, and Sanja Fidler. Extracting triangular 3d models, materials, and lighting from images. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8280–8290, 2022. 2

[30] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008. 5

[31] Baptiste Nicolet, Alec Jacobson, and Wenzel Jakob. Large steps in inverse rendering of geometry. *ACM Transactions on Graphics (TOG)*, 40(6):1–13, 2021. 2

[32] Michael Oechsle, Songyou Peng, and Andreas Geiger. Unisurf: Unifying neural implicit surfaces and radiance fields for multi-view reconstruction. In *International Conference on Computer Vision (ICCV)*, 2021. 2

[33] Werner Palfinger. Continuous remeshing for inverse rendering. *Computer Animation and Virtual Worlds*, 33(5):e2101, 2022. 2, 7, 6

[34] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 165–174, 2019. 2

[35] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. 5

[36] Bui Tuong Phong. Illumination for computer generated pictures. In *Seminal graphics: pioneering efforts that shaped the field*, pages 95–101. 1998. 7

[37] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. Accelerating 3d deep learning with pytorch3d. *arXiv preprint arXiv:2007.08501*, 2020. 4

[38] Tianchang Shen, Jun Gao, Kangxue Yin, Ming-Yu Liu, and Sanja Fidler. Deep marching tetrahedra: a hybrid representation for high-resolution 3d shape synthesis. *Advances in Neural Information Processing Systems*, 34:6087–6101, 2021. 2, 7, 6

[39] Tianchang Shen, Jacob Munkberg, Jon Hasselgren, Kangxue Yin, Zian Wang, Wenzheng Chen, Zan Gojcic, Sanja Fidler, Nicholas Sharp, and Jun Gao. Flexible isosurface extraction for gradient-based mesh optimization. *ACM Transactions on Graphics (TOG)*, 42(4):1–16, 2023. 2, 7, 6

[40] Tianchang Shen, Zhaoshuo Li, Marc Law, Matan Atzmon, Sanja Fidler, James Lucas, Jun Gao, and Nicholas Sharp. Spacemesh: A continuous representation for learning manifold surface meshes. *arXiv preprint arXiv:2409.20562*, 2024. 1, 2

[41] Yawar Siddiqui, Antonio Alliegro, Alexey Artemov, Tatiana Tommasi, Daniele Sirigatti, Vladislav Rosov, Angela Dai, and Matthias Nießner. Meshgpt: Generating triangle meshes with decoder-only transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 19615–19625, 2024. 1, 2

[42] Sanghyun Son, Matheus Gadelha, Yang Zhou, Zexiang Xu, Ming C Lin, and Yi Zhou. Dmesh: A differentiable representation for general meshes. *arXiv preprint arXiv:2404.13445*, 2024. 1, 2, 3, 4, 5, 6, 7, 8, 10

[43] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9 (11), 2008. 2

[44] Peng Wang, Lingjie Liu, Yuan Liu, Christian Theobalt, Taku Komura, and Wenping Wang. Neus: Learning neural implicit surfaces by volume rendering for multi-view reconstruction. *arXiv preprint arXiv:2106.10689*, 2021. 2

[45] Yiming Wang, Qin Han, Marc Habermann, Kostas Daniilidis, Christian Theobalt, and Lingjie Liu. Neus2: Fast learning of neural implicit surfaces for multi-view reconstruction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023.

[46] Xinyue Wei, Fanbo Xiang, Sai Bi, Anpei Chen, Kalyan Sunkavalli, Zexiang Xu, and Hao Su. Neumanifold: Neural watertight manifold reconstruction with efficient and high-quality rendering support. *arXiv preprint arXiv:2305.17134*, 2023. 2

[47] Xinyue Wei, Kai Zhang, Sai Bi, Hao Tan, Fujun Luan, Valentin Deschaintre, Kalyan Sunkavalli, Hao Su, and Zexiang Xu. Meshlrm: Large reconstruction model for high-quality mesh. *arXiv preprint arXiv:2404.12385*, 2024. 2

[48] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992. 9, 10

[49] Lior Yariv, Jiatao Gu, Yoni Kasten, and Yaron Lipman. Volume rendering of neural implicit surfaces. In *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021. 2

[50] Zhengming Yu, Zhiyang Dou, Xiaoxiao Long, Cheng Lin, Zekun Li, Yuan Liu, Norman Müller, Taku Komura, Marc Habermann, Christian Theobalt, et al. Surf-d: High-quality surface generation for arbitrary topologies using diffusion models. *arXiv preprint arXiv:2311.17050*, 2023. 2

[51] Longwen Zhang, Ziyu Wang, Qixuan Zhang, Qiwei Qiu, Anqi Pang, Haoran Jiang, Wei Yang, Lan Xu, and Jingyi Yu. Clay: A controllable large-scale generative model for creating high-quality 3d assets. *ACM Transactions on Graphics (TOG)*, 43(4):1–20, 2024. 2

[52] Qingnan Zhou and Alec Jacobson. Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797*, 2016. 2, 5, 6, 7

[53] Yi Zhou, Chenglei Wu, Zimo Li, Chen Cao, Yuting Ye, Jason Saragih, Hao Li, and Yaser Sheikh. Fully convolutional mesh autoencoder using efficient spatially varying kernels. *Advances in neural information processing systems*, 33:9251–9262, 2020. 2

DMesh++: An Efficient Differentiable Mesh for Complex Shapes

Supplementary Material

7. Details about *Minimum-Ball* algorithm

7.1. Algorithm

Algorithm 1 `Minimum-Ball`

1: $\mathbb{P}, \mathbb{F} \leftarrow$ Set of points and query faces
2: $\alpha_{min} \leftarrow$ Coefficient for sigmoid function
3: $B^c_{\mathbb{F}}, B^r_{\mathbb{F}} \leftarrow$ `Compute-Minimum-Ball` $(\mathbb{P}, \mathbb{F})$
4: $P^{nearest}_{\mathbb{F}} \leftarrow$ `Find-Nearest-Neighbor` $(B^c_{\mathbb{F}}, \mathbb{P})$
5: $d(B_{\mathbb{F}}, \mathbb{P}) \leftarrow B^r_{\mathbb{F}} - ||P^{nearest}_{\mathbb{F}} - B^c_{\mathbb{F}}||$
6: $\lambda_{min}(\mathbb{F}) \leftarrow \sigma(d(B_{\mathbb{F}}, \mathbb{P}) \cdot \alpha_{min})$
7: **return** $\lambda_{min}(\mathbb{F})$

We formally describe the *Minimum-Ball algorithm* in Algorithm 1.

- **Line 1:** We define the given set of points (specifically, their positions) as $\mathbb{P}$ and the query faces as $\mathbb{F}$.
- **Line 2:** We introduce α_{min}, the coefficient for the sigmoid function used to map the signed distance to a probability. Details on determining α_{min} are provided in Appendix 7.3.
- **Line 3:** For each query face $F \in \mathbb{F}$, we compute the minimum bounding ball (B_F) as described in Appendix 7.2. We denote the entire set of bounding balls as $B_{\mathbb{F}}$, their centers as $B^c_{\mathbb{F}}$, and their radii as $B^r_{\mathbb{F}}$.
- **Line 4:** For each $F \in \mathbb{F}$, we find the nearest neighbor of B^c_F in $\mathbb{P} - F$. However, this operation cannot be parallelized across all query faces because the set $\mathbb{P} - F$ varies for each face. To address this, we find $(d+1)$-nearest neighbors of B^c_F in $\mathbb{P}$, where d is the spatial dimension. This approach ensures correctness in two scenarios:
  - If $F \in \mathbb{F}_{min}$, the bounding ball B_F does not contain any points from $\mathbb{P}$ within it, and the points on F are the d-nearest neighbors of B^c_F. To find the nearest neighbor in $\mathbb{P}-F$, we need to consider $(d+1)$-nearest neighbors.
  - If $F \notin \mathbb{F}_{min}$, only the single nearest neighbor of B^c_F is relevant.

  To safely handle both cases, we always search for $(d+1)$-nearest neighbors and then select the first neighbor from the list that does not belong to F.
- **Line 5:** We compute the signed distance $d(B_{\mathbb{F}}, \mathbb{P})$ for all query faces.
- **Line 6:** The signed distance is converted to a probability using the sigmoid function, with α_{min} as the scaling factor.
- **Line 7:** Finally, the algorithm returns the computed probabilities for all faces.

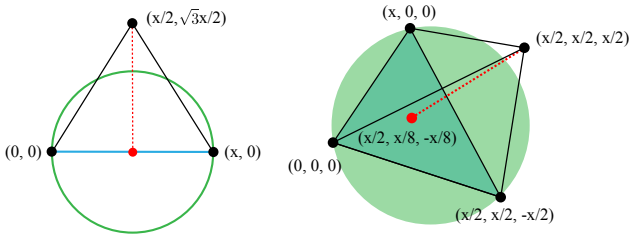


Figure 13. **Common signed distance for a 2D (left) and 3D (right) face in (initial) regular grid.** We compute the signed distance by subtracting the radius of the minimum bounding ball from the length of the red line. The red dot represents the center of the minimum bounding ball.

7.2. Minimum-Ball computation

Let us define a face $F = \{p_1, p_2, \ldots, p_d\}$, where $p_i \in \mathbb{P}$. To determine the bounding balls of F, we first identify the set of points that are equidistant from the vertices of F. Among these, we select the point lying on the hyperplane containing F as the center of the minimum bounding ball, denoted as B^c_F.

When $d = 2$, the center simplifies to the midpoint of F:

$$B^c_F\big|_{d=2} = \frac{1}{2}(p_1 + p_2). \tag{6}$$

For $d = 3$, the computation is more complex [7]:

$$B^c_F\big|_{d=3} = p_1 + \frac{||d_2||^2(d_1 \times d_2) \times d_1 + ||d_1||^2(d_2 \times d_1) \times d_2}{2||d_1 \times d_2||^2}, \tag{7}$$

where $d_1 = p_2 - p_1$ and $d_2 = p_3 - p_1$.

Unlike the case where $d = 2$, for $d = 3$, we cannot compute B^c_F if $||d_1 \times d_2|| = 0$. During computation, cases where this value falls below a certain threshold are marked and excluded from subsequent steps to avoid numerical instability.

After determining B^c_F, we calculate the radius B^r_F as the distance between B^c_F and the points on F.

7.3. Sigmoid coefficient α_{min}

The sigmoid coefficient α_{min} plays a critical role in determining the probability to which a signed distance d is mapped. Even if a face F satisfies the *Minimum-Ball* condition by a large margin ($d(B_F, \mathbb{F}) \gg 0$), indicating a high existence probability for F, a small α_{min} value would result in the derived probability being only slightly greater

[7] Derived from the Geometry Junkyard: https://ics.uci.edu/~eppstein/junkyard/circumcenter.html

than 0.5. To minimize such mismatches, we set α_{min} based on the density of the grid from which optimization begins.

As discussed in Appendix 8.2.1, the reconstruction process often starts from a fixed triangular (2D) or tetrahedral (3D) grid (Fig. 16). At the initial state, every face in the grid satisfies the *Minimum-Ball* condition (Appendix 8.2.1). Notably, every interior face in the grid shares a common signed distance $d_{common} > 0$. Let us denote x as the edge length of the grid, applicable for both 2D and 3D cases. Then, the common signed distance can be computed as follows:

For $d = 2$:

$$d_{common} = \frac{\sqrt{3}-1}{2}x. \tag{8}$$

For $d = 3$:

$$d_{common} = \frac{\sqrt{34}-3\sqrt{2}}{8}x. \tag{9}$$

In Fig. 13, we provide an illustration of the reasoning behind these results. By calculating these common signed distances, we use them to determine α_{min}. Specifically, during the first epoch, we set $\alpha_{min} = 32/d_{common}$, ensuring that the probability for every face in the grid is initialized to $\sigma(32) \simeq 1.0$.

In subsequent epochs, α_{min} is adjusted to account for the additional points introduced during subdivision. If α^1_{min} represents the value in the first epoch, the value for the i-th epoch is given by:

$$\alpha^i_{min} = \frac{\alpha^1_{min}}{2^{i-1}}. \tag{10}$$

7.4. Nearest neighbor caching

In Secs. 3.2 and 5.1, we demonstrated how the *Minimum-Ball* algorithm significantly accelerates tessellation. This process can be further optimized by periodically caching the K-nearest neighbors of each B^c_F in $\mathbb{P}$ and using the cached neighbors for computing probabilities until the next cache update. This optimization is feasible because the K-nearest neighbors generally do not change significantly during the optimization process.

Let us define the number of optimization steps as n_0 and the number of steps between cache updates as n_1. At every n_1 steps, we refresh the query faces $\mathbb{F}$ based on the current set of points $\mathbb{P}$ and recompute the centers of the minimum bounding balls for the query faces ($B^c_{\mathbb{F}}$). Then, we identify the K-nearest neighbors of $B^c_{\mathbb{F}}$ in $\mathbb{P}$. In practice, we compute the $(K + d)$-nearest neighbors instead, as explained in Appendix 7.1, to ensure robustness.

During the subsequent optimization steps, for a given face F, we compute the distance from B^c_F to the cached K-nearest neighbors in $\mathbb{P}$ and select the nearest neighbor from the cache to compute the signed distance $d(B_F, \mathbb{P})$.

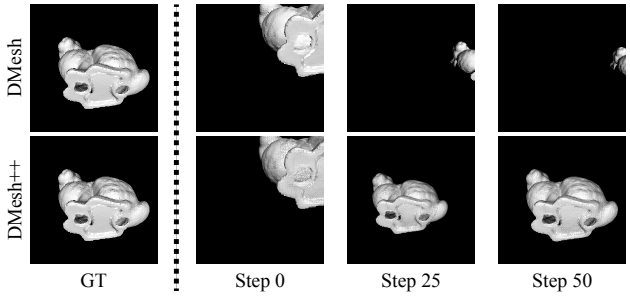


Figure 14. **Role of visibility gradient in geometric optimization.** In this experiment, we optimize the translation vector of the object by comparing its rendered image and the ground truth image on the left. Since the differentiable renderer of DMesh [42] does not implement visibility gradient, while ours does, DMesh fails to find the correct translation vector.

This mechanism is described in detail in Algorithm 2 and Appendix 8.2.2 in the context of point position optimization. In our experiments for 3D multi-view reconstruction, we set $n_0 = 2000$, $n_1 = 50$, and $K = 10$.

8. Details about Reconstruction Process

In this section, we provide implementation details about our reconstruction process described in Sec. 4. Before delving into these details, we introduce the loss formulations for reconstruction problems.

8.1. Loss Formulation

Our final loss, L, is comprised of main reconstruction loss (L_{recon}) and two regularization terms: L_{qual} and L_{real}.

$$L = L_{recon} + \lambda_{qual} \cdot L_{qual} + \lambda_{real} \cdot L_{real}. \tag{11}$$

We explain each of these terms below.

8.1.1. Reconstruction Loss (L_{recon})

Reconstruction loss drives the reconstruction process by comparing our current probabilistic mesh and the given ground truth observations. For different observations, we need different loss functions as follows.

Point Cloud When ground truth point clouds are provided, we utilize the expected Chamfer Distance (CD) proposed by [42]. In this formulation, when sampling points from our mesh, we assign an existence probability to each sampled point, which matches the probability of the face from which the point is sampled. The expected CD incorporates these probabilities, unlike the traditional Chamfer Distance, which does not. For further details, refer to [42].

Multi-view Images For rendering probabilistic faces, we interpret each face's existence probability as its opacity, following [42]. To render large number of semi-transparent

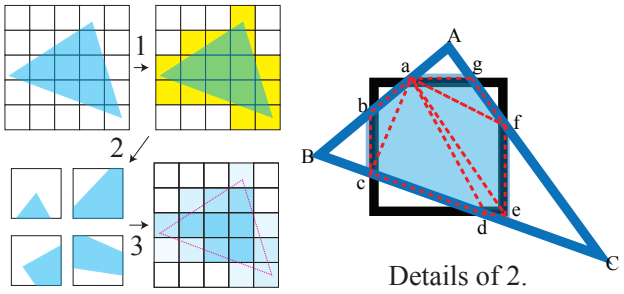


Figure 15. **Implementation of anti-aliasing in our differentiable renderer.** On the left, we show the process of anti-aliasing: 1) Find pixels that overlap with the given triangle, 2) Find the area that each pixel overlaps with the given triangle, and 3) Determine the color of each pixel based on the overlap area. On the right, we show details of the step 2.

faces efficiently, we use the differentiable renderer of [42], but we found out that it does not implement visibility gradient that is necessary for optimizing geometric properties (Fig. 14). For the details about this visibility gradient, please refer to [19], which implemented the visibility gradient using anti-aliasing. Following their path, we enhanced the differentiable renderer of DMesh by implementing anti-aliasing in CUDA, which provides us visibility gradients.

In Fig. 15, we briefly illustrate how we implemented anti-aliasing in CUDA. Specifically, for each (triangular) face-pixel pair (F, P), we project F onto the image space and compute the overlapping area $A(F, P)$ between the projected triangle and the pixel (Fig. 15 right, blue area). Denoting the total area of the pixel as $A(P)$, the ratio of the overlapping area in the given pixel, $\rho(F, P)$, is computed as:

$$\rho(F, P) = \frac{A(F, P)}{A(P)}. \tag{12}$$

We use $\rho(F, P)$ to determine the opacity of the face F at the pixel P. If the opacity of F is $\alpha(F)$, we compute the face opacity at pixel P, $\alpha(F, P)$, as:

$$\alpha(F, P) = \alpha(F) \cdot \rho(F, P) \leq \alpha(F). \tag{13}$$

Thus, the opacity of F at P is proportional to the overlapping area between the triangle and the pixel.

In the right figure of Fig. 15, we illustrate the process of computing the overlapping area $A(F, P)$. The vertices of F are projected onto the image plane and visited in counter-clockwise order (e.g., A - B - C - A in the illustration). We then find the intersection points between the triangle edges and the pixel boundaries. These intersection points form the vertices of the (convex) overlapping polygon.

For example, vertices (a) and (b) are found by calculating the intersections of $\overline{AB}$ with the pixel boundaries. The vertices of the overlapping polygon are stored in counter-clockwise order, and the polygon is subdivided into a set of sub-triangles, as shown by the dotted red lines in the visualization. The total area of the overlapping polygon is obtained by summing the areas of the sub-triangles.

Using this enhanced differentiable renderer, we render multi-view images and compute the L_1 loss between the ground truth images as the reconstruction loss, L_{recon}.

8.1.2. Triangle Quality Loss (L_{qual})

To improve the triangle quality of the final mesh, we adopt the triangle quality loss (L_{qual}) of DMesh [42]. Specifically, the loss is defined as follows:

$$L_{qual} = \frac{1}{|\mathbb{F}|} \sum_{F \in \mathbb{F}} AR(F) \cdot \Lambda(F), \tag{14}$$

where $\mathbb{F}$ is the set of every face combination we consider, $AR(\cdot)$ is a function that computes aspect ratio of the given face, and $\Lambda(\cdot)$ is the face probability function defined in Sec. 3.1.

8.1.3. Real Loss (L_{real})

We minimize the sum of point-wise real values (ψ), so that we can remove redundant faces as much as possible during optimization. The loss L_{real} is simply defined as:

$$L_{real} = \frac{1}{|\mathbb{P}|} \sum_{p \in \mathbb{P}} \Psi(p), \tag{15}$$

where $\Psi(\cdot)$ is the function that returns the real value of the given point, as defined in Sec. 3.1.

8.2. Reconstruction Steps

Now we provide detailed explanations about each step in the reconstruction process.

8.2.1. Step 1: Initialization

We initialize our point features differently for two different scenarios: when sample point cloud is given or not.

Point Cloud Init. When a point cloud sampled from the target shape is given, we can initialize our point features using the point cloud, so that the initial configuration already captures the overall structure of the target shape (Fig. 3). Specifically, for the given point cloud, we estimate the density of the point cloud by computing the distance to the nearest point for each point. Then, we down sample the point cloud using a voxel grid, of which size is defined as the point cloud density, to remove redundant points and holes. Finally, we follow the initialization scheme of DMesh [42] using the down sampled point cloud.

Regular Grid Init. If we do not have any prior knowledge about the target shape, we first organize the points in a regular grid, ensuring that every face in the grid satisfies

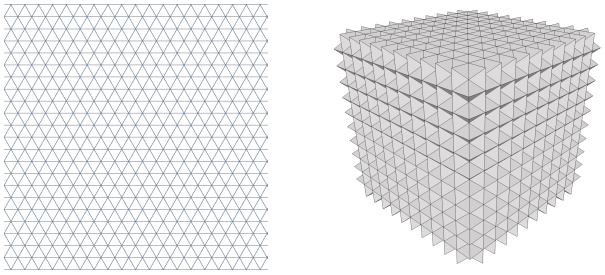


Figure 16. **Grid structure to initialize real values in 2D (left) and 3D (right).** Every face in the grid structure satisfies *Minimum-Ball* condition (Definition 3.1).

the *Minimum-Ball* condition (Definition 3.1), and initialize the point-wise real values (ψ) with additional features (*e.g.* colors). This regular grid guarantees that the faces observed in this step will also be observable in the subsequent step, where the *Minimum-Ball* algorithm determines face existence. For $d = 2$, this condition is satisfied by forming every triangle in the grid as an equilateral triangle. For $d = 3$, we use a body-centered cubic lattice. The grids are illustrated in Fig. 16.

With these fixed points and faces, we formulate the final loss by setting $\lambda_{qual} = 0$ and $\lambda_{real} = 10^{-4}$ in Eq. (11), and minimize it to determine which faces to include in the final mesh. After optimization, we collect points with real values larger than 0.01 to ensure that as many faces as possible are available for the next optimization step, thereby reducing the risk of holes in the surface.

8.2.2. Step 2: Position Optimization

In this step, we fix the point-wise real values (ψ) and optimize only the point positions. For clarity, we formally describe the process in Algorithm 2, and explain the algorithm line by line below:

- **Lines 1-2:** For the given set of points, we denote their positions as $\mathbb{P}$ and their real values as Ψ.
- **Line 3:** We define the total number of optimization steps as n_0.
- **Line 4:** We define the number of optimization steps required to refresh query faces and their nearest neighbor cache as n_1. Since point positions are optimized, the point configuration evolves during optimization, potentially leading to the emergence of new faces that were previously unobservable. To account for these changes, we refresh the query faces periodically.
- **Line 5:** We denote the number of nearest neighbors to store in the cache for the query faces as K.
- **Lines 6-7:** The optimization process runs for n_0 steps.
- **Lines 8-12:** At every n_1 step, we update the query faces based on the current point configuration.
  In the `Update-Query-Faces` function, which uses point positions and their real values, we:
  - Extract points with a real value of 1.
  - For each extracted point, find its 10-nearest neighbors that also have a real value of 1, since any face containing a point with a real value of 0 is considered non-existent.
  - Perform Delaunay Triangulation (DT) for the entire point set and collect faces in DT where all points have a real value of 1. This ensures the inclusion of as many faces as possible during optimization, helping to eliminate potential holes later.

  For the updated query faces, we compute the centers of their minimum bounding balls. Subsequently, we identify the K-nearest neighbors of these centers in $\mathbb{P}$ and store this information in the nearest neighbor cache $\mathbb{C}$.
- **Lines 13-16:** Using the current point configuration, we compute the minimum bounding balls ($B_{\mathbb{F}}$) for the query faces. For each bounding ball center, we find the nearest neighbor in the nearest neighbor cache $\mathbb{C}$ by calculating the distances to points in $\mathbb{C}$ and selecting the closest one. We then compute the signed distance $d(B_{\mathbb{F}}, \mathbb{P})$ for the query faces and use it to get the probability $\lambda_{min}(\mathbb{F})$.
- **Line 17:** Since the query faces consist only of points with a real value of 1, we set the final face probability $\lambda(\mathbb{F})$ to be the same as $\lambda_{min}(\mathbb{F})$ (Sec. 3.1).
- **Line 18:** Based on the point positions, query faces, and their existence probabilities, we compute the loss L to minimize following Eq. (11).
- **Lines 19-20:** Finally, we update the point positions $\mathbb{P}$ to minimize L and iterate the process.

Algorithm 2 `Position Optimization`

1: $\mathbb{P}, \Psi \leftarrow$ Set of points and their real values
2: $\alpha_{min} \leftarrow$ Coefficient for sigmoid function
3: $n_0 \leftarrow$ Number of optimization steps
4: $n_1 \leftarrow$ Number of refresh steps for query faces
5: $K \leftarrow$ Number of nearest neighbors to store in cache
6: $i \leftarrow 0$
7: **while** $i < n_0$ **do**
8: **if** $i \mod n_1 = 0$ **then**
9: $\mathbb{F} \leftarrow$ `Update-Query-Faces`$(\mathbb{P}, \Psi)$
10: $B^c_{\mathbb{F}}, B^r_{\mathbb{F}} \leftarrow$ `Compute-Minimum-Ball`$(\mathbb{P}, \mathbb{F})$
11: $\mathbb{C} \leftarrow$ `Find-KNN`$(B^c_{\mathbb{F}}, \mathbb{P}, K)$
12: **end if**
13: $B^c_{\mathbb{F}}, B^r_{\mathbb{F}} \leftarrow$ `Compute-Minimum-Ball`$(\mathbb{P}, \mathbb{F})$
14: $P^{nearest}_{\mathbb{F}} \leftarrow$ `Find-NN-CACHE`$(B^c_{\mathbb{F}}, \mathbb{C})$
15: $d(B_{\mathbb{F}}, \mathbb{P}) \leftarrow B^r_{\mathbb{F}} - ||P^{nearest}_{\mathbb{F}} - B^c_{\mathbb{F}}||$
16: $\lambda_{min}(\mathbb{F}) \leftarrow \sigma(d(B_{\mathbb{F}}, \mathbb{P}) \cdot \alpha_{min})$
17: $\lambda(\mathbb{F}) \leftarrow \lambda_{min}(\mathbb{F})$
18: $L \leftarrow$ `Compute-Loss`$(\mathbb{P}, \mathbb{F}, \lambda(\mathbb{F}))$
19: Update $\mathbb{P}$ to minimize L
20: $i \leftarrow i + 1$
21: **end while**

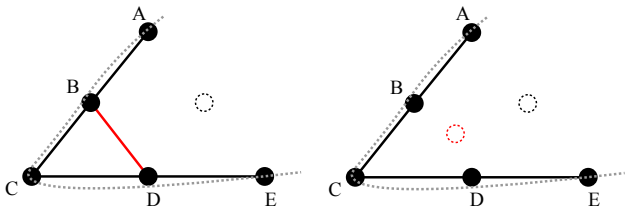


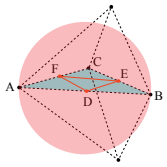
Figure 17. **Point insertion for removing undesirable face.** (Left) To reconstruct the ground truth shape, we need to set the real value (ψ) of points A-E to 1. The point rendered with dotted line has real value of 0. Then, we observe unnecessary face $\overline{BD}$ exists. (Right) To remove this face, we insert additional point that carries 0 real value near the unnecessary face.

8.2.3. Step 3: Real Value Optimization

In this step, we re-optimize the point-wise real values while keeping the point positions fixed. From the current point configuration, we identify all faces in the Delaunay Triangulation (DT) that satisfy the *Minimum-Ball* condition. Note that any face satisfying this condition must exist in the DT (Lemma 3.2). Thus, we first compute the DT of the points and then verify whether each face in the DT satisfies the *Minimum-Ball* condition.

Next, we follow a similar optimization process to Step 1 (Appendix 8.2.1). Additionally, if it was the multi-view reconstruction task, we remove invisible faces to remove redundant faces as much as possible. If this was the last epoch, we return the post-processed mesh.

8.2.4. Step 4: Subdivision



To reconstruct fine geometric details of the given shape, we subdivide the current mesh mainly by adding points with $\psi = 1$ at the middle of edges that are adjacent to currently existing faces. In the inset, points E, D, F are the newly inserted points. They form 4 sub-faces with A, B, C, and they all satisfy Definition 3.1. Therefore, we can guarantee that these sub-faces will exist at the start of next epoch. Note that this guarantee does not hold for WDT.

At the same time, it is also possible to insert additional points into faces that *should not* exist in the next epoch, effectively removing such faces at the start of the next epoch. For example, during the real value optimization step in the pipeline (Fig. 6, Appendix 8.2.3), invisible faces are removed for multi-view reconstruction task. After optimization, we may observe removed faces with all their points have a real value of 1.0, creating a contradiction. This situation could arise due to ambiguities in the mesh definition, as illustrated in Fig. 17.

To eliminate these undesirable faces, additional points with a real value of 0 are inserted at their circumcenters. Consequently, after subdivision, several holes may appear on the surface because these additional points might also be inserted into faces that *should* exist (Fig. 6). However, most of these holes are resolved during subsequent optimization steps.

9. Experimental Details and Additional Results

In this section, we outline the experimental settings used for the results in Sec. 5 and present additional results to support our claims.

9.1. Dataset

Here, we provide details on the datasets described in Sec. 5.2.

9.1.1. Font

We used four font styles: Pacifico, Permanent-Marker, Playfair-Display, and Roboto.

9.1.2. Thingi10K

We manually selected 10 closed surfaces and 10 open surfaces from the Thingi10K dataset [52]. Specifically, we used the following models, denoted by their file IDs:

- **Closed surfaces:** 47926, 68380, 75147, 80650, 98576, 101582, 135730, 274379, 331105, and 372055.
- **Open surfaces:** 40009, 41909, 73058, 82541, 85538, 131487, 75846, 76278, 73421, and 106619.

These models were chosen because they exhibit minimal self-occlusions, enabling dense observations from multi-view images. Additionally, we randomly selected 500 models and used for comparisons.

9.1.3. Objaverse

We manually selected 30 mesh models that exhibit diverse topology from Objaverse [10], which include both closed and open surfaces, and also have small scenes. Some of these models are rendered in Figs. 3, 6, 9, 19 and 20.

9.2. 2D Point Cloud Reconstruction

9.2.1. Hyperparameters

DMesh++

- Learning rate (real value, ψ): 0.3
- Learning rate (position): 0.001
- Number of epochs: 1
- Number of optimization steps
  - Step 1 (Real value initialization): 100
  - Step 2 (Point position optimization): 500

9.2.2. Reconstruction of Complex Drawings

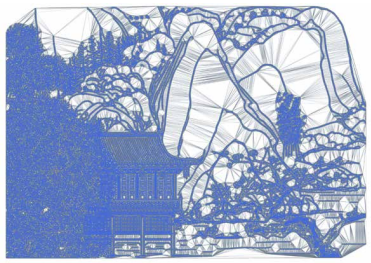
In Fig. 18, we present the reconstruction results for complex 2D drawings. As the figure illustrates, DMesh++ successfully reconstructs intricate 2D geometries from point clouds, even when the number of edges approaches nearly 1 million.

(a) Flower, # Edge = 99K, 6 min.

(b) Eagle, # Edge = 179K, 11 min.

(c) Picasso, # Edge = 159K, 8 min.

(d) Egyptian, # Edge = 227K, 19 min.



(e) Chinese, # Edge = 987K, 86 min.

Figure 18. **2D point cloud reconstruction result for complex drawings.** For each drawing, we report both the number of edges and the reconstruction time. For the Chinese drawing, we additionally render the "imaginary" part on the right to clearly illustrate its complexity.

9.3. 3D Point Cloud Reconstruction

9.3.1. Hyperparameters

DMesh++

- Initial Grid Edge Length: 3× input point cloud density
- Learning rate (position): 0.001
- Number of epochs: 1
- Number of optimization steps
  - Step 2 (Point position optimization): 2000
  - Step 3 (Real value optimization): 0

9.4. 3D Multi-View Reconstruction

9.4.1. Hyperparameters

Remeshing [33]

- Image Batch Size: 8
- Number of Optimization Steps: 1000
- Learning Rate: 0.1
- Edge Length Limits: [0.02, 0.15]

The "Edge Length Limits" were adjusted to produce meshes with a similar number of vertices and faces to other methods for a fair comparison.

DMTet [38]

- Image Batch Size: 8
- Number of Optimization Steps: 5000
- Learning Rate: 0.001
- Grid Resolution: 128

The SDF was initialized to a sphere, as in the original implementation, before starting optimization.

FlexiCubes [39]

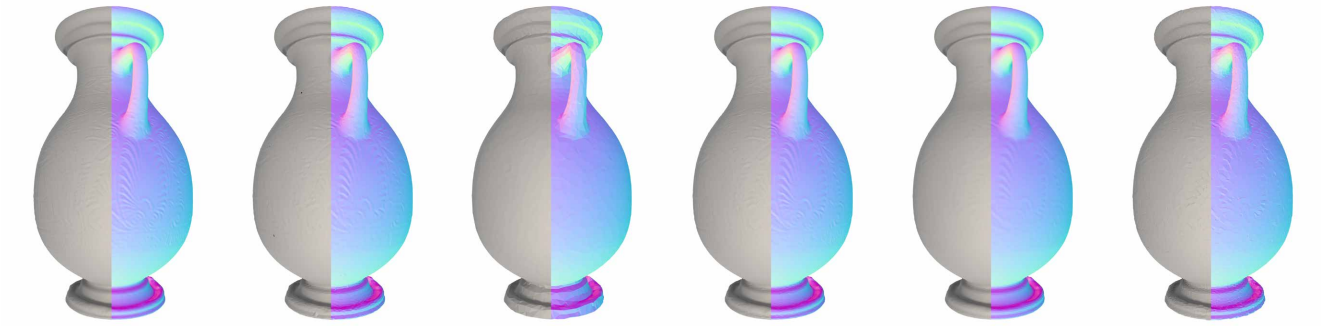
- Image Batch Size: 8
- Number of Optimization Steps: 2000
- Number of Warm-up Steps: 1500
- Learning Rate: 0.01
- Grid Resolution: 80
- Triangle Aspect Ratio Loss Weight: 0.01

The SDF was initialized randomly, following the original implementation. To improve the quality of the output mesh, we adopted a triangle aspect ratio loss, designed to minimize the average aspect ratio of triangles in the mesh. The mesh was first optimized for 1500 steps as a warm-up without the triangle aspect ratio loss, followed by 500 steps with the additional loss.

Additionally, we observed that the output mesh often included false internal structures, which significantly degraded the Chamfer Distance (CD) compared to the ground truth mesh. To mitigate this, we performed a visibility test on the output mesh to remove these false internal structures as much as possible.

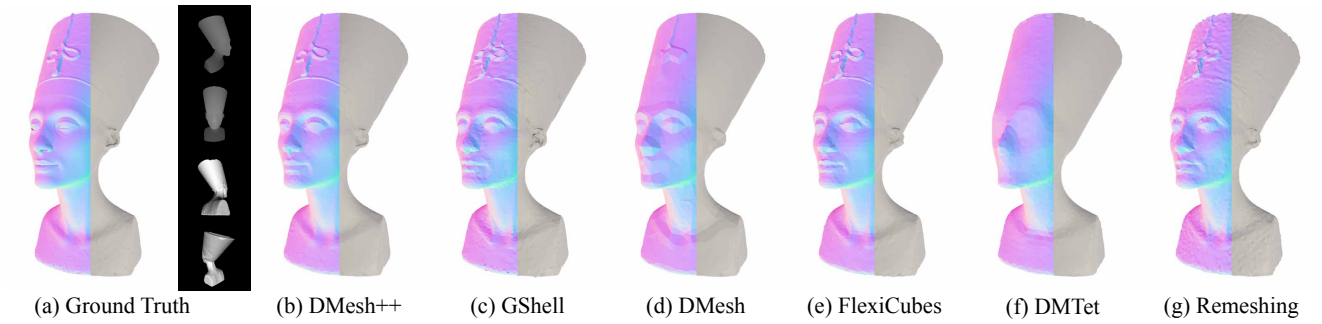
GShell [23]

- Image Batch Size: 8
- Number of Optimization Steps: 5000
- Number of Warm-up Steps: 4500



(a) Ground Truth (b) DMesh++ (c) DMesh (d) PoNQ (e) PSR (f) VoroMesh

Figure 19. **Qualitative comparison of 3D point cloud reconstruction results for a closed surface (vase).** For each image, we render the view-point normal on the right, and the diffuse image on the left. Among the baseline methods that reconstruct watertight mesh from point clouds, PoNQ [27] performs the best in reconstructing fine geometric details. While DMesh [42] fails at reconstructing such details due to the lack of mesh complexity, DMesh++ successfully recovers them and produce comparable result to PoNQ.



(a) Ground Truth (b) DMesh++ (c) GShell (d) DMesh (e) FlexiCubes (f) DMTet (g) Remeshing

Figure 20. **Qualitative comparison of 3D multi-view reconstruction results for a closed surface (sculpture).** We render the input diffuse and depth images alongside the ground truth image. For each image, we render the view-point normal on the left, and the diffuse image on the right. We can observe that the reconstruction result of DMesh++ is as good as the other baseline methods that are optimized for closed surfaces.

- Learning Rate: 0.01
- Grid Resolution: 80
- Triangle Aspect Ratio Loss Weight: 0.0001

To enhance the quality of the output mesh, we employed the same additional measures as FlexiCubes. We found that longer optimization steps were required for GShell compared to FlexiCubes to effectively handle open surfaces.

DMesh++ Settings

- Initial Grid Edge Length: 0.05
- Learning Rate (Real Value, ψ): 0.01
- Learning Rate (Position): 0.001
- Number of Epochs: 2
  - Image Res. / Batch Size at Epoch 1: (256, 256), 1
  - Image Res. / Batch Size at Epoch 2: (512, 512), 1
- Number of Optimization Steps
  - Step 1 (Real Value Initialization): 1000
  - Step 2 (Point Position Optimization): 2000
  - Step 3 (Real Value Optimization): 1000

In the first epoch, we used lower-resolution images as part of a coarse-to-fine approach.

9.4.2. Limitations

Despite DMesh++'s success in reconstructing geometrically accurate meshes from multi-view images (of a synthetic object or scene), it currently cannot recover meshes from real-world images. This limitation stems from an inadequate rendering model — our current per-vertex color model is too simple to capture detailed geometry, and our algorithm assumes full knowledge of lighting conditions, which is not available in real-world scenarios.

To illustrate this, we applied our reconstruction algorithm to real-world images from the DTU dataset [15], as shown in Fig. 21. While our method approximates the real-world images (Fig. 21(b)), the extracted mesh exhibits numerous false floaters (Fig. 21(c)). We believe this suboptimal result is due to the lack of proper rendering models and regularizations, and addressing this issue by integrating DMesh++ with other reconstruction mechanisms is an exciting direction for future research, as discussed in Sec. 6.

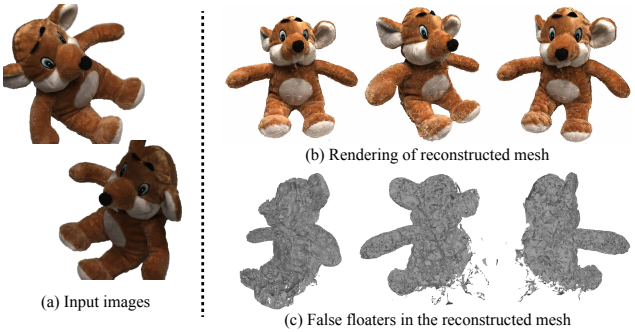


Figure 21. **3D reconstruction from real-world images in DTU dataset [15].** The input images are shown on left, and the reconstructed mesh is shown on right.

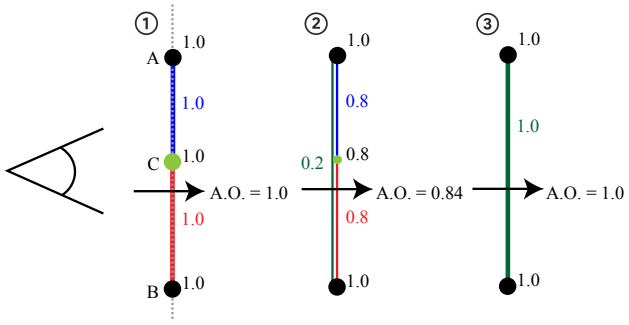


Figure 22. **Local minima of weight regularization in a rendering setting.** (1) The ground truth geometry is rendered in gray dotted line. There are 3 points (A, B, C), where only the end points (A, B) are necessary for fully representing the underlying shape. Every point has weight 1.0, which is written to the next of each point. In this case, faces $\overline{AC}$ and $\overline{BC}$ exist with probability 1.0, which corresponds to their opacity. In this case, for a ray that goes through this mesh, the accumulated opacity (A.O.) becomes 1.0, and the reconstruction loss is 0. (2) When weight regularization reduces the weight of (redundant) C to 0.8, the probability of faces $\overline{AC}$ and $\overline{BC}$ becomes 0.8, and that of $\overline{AB}$ becomes 0.2. However, in this case, the accumulated opacity of the same ray becomes 0.84, which results in non-zero reconstruction loss. (3) Therefore, with a small weight regularization, we cannot remove C to get this optimal mesh, which contains only $\overline{AB}$, and attains 0 reconstruction loss.

10. *Reinforce-Ball* algorithm

Here we introduce an experimental algorithm that further enhances DMesh++'s capability. As discussed in Sec. 3.1, DMesh++ no longer uses the per-point weights found in DMesh [42]. In DMesh, optimizing per-point weights helps control mesh complexity: stronger regularization on these weights results in a simpler output mesh. Since DMesh++ lacks this mechanism, it cannot directly regulate mesh complexity during optimization. To address this limitation, we propose the *Reinforce-Ball* algorithm, which reduces unnecessary faces while preserving essential geometric details.

10.1. Local Minima of Weight Regularization

Before delving into the details of the Reinforce-Ball algorithm, we first highlight a limitation of DMesh's per-point weight regularization. Specifically, while per-point weights are relevant for controlling mesh complexity, they alone cannot achieve adaptive resolution or produce a mesh that is both efficient and precise. Below, we explain the reasons in detail, assuming that per-point probabilities are optimized and that the *Minimum-Ball* condition is employed to compute face probabilities.

In Fig. 22, we provide an example in a rendering scenario. A camera is placed on the left, and three different probabilistic meshes are shown on the right.

In **case (1)**, there are three points: A, B, and C. By connecting points A and B, the ground truth shape can be perfectly reconstructed, making point C redundant. Assume the optimization starts from this state, where all points have an existence probability of 1.0. According to the *Minimum-Ball* condition, the probabilities of faces $\overline{AC}$ and $\overline{BC}$ will also be 1.0. In this scenario, if a ray from the camera intersects the mesh, the accumulated opacity will be 1.0, representing a fully opaque surface. Consequently, the reconstruction loss will be 0.0, as the fully opaque faces perfectly match the ground truth.

In **case (3)**, the optimal configuration is rendered, where the redundant point C is removed. The probability of face $\overline{AB}$ becomes 1.0, making it fully opaque. Again, the reconstruction loss is 0.0.

In **case (2)**, an intermediate state between cases (1) and (3) is rendered. Assume that the probability of point C is reduced to 0.8 due to regularization. Consequently, the probabilities of faces $\overline{AC}$ and $\overline{BC}$ are also reduced to 0.8 because one of their endpoints, C, has a probability of 0.8. Simultaneously, the probability of face $\overline{AB}$ increases from 0 to 0.2, as the probability of point C, which lies inside the minimum bounding ball of the face, is 0.8.

Now, consider a camera ray passing through $\overline{AB}$ and $\overline{BC}$ sequentially (the order does not matter due to their tight overlap). Using alpha blending, the accumulated opacity is computed as:

$$\text{Accumulated Opacity: } 0.2 + (1.0 - 0.2) \cdot 0.8 = 0.84. \quad (16)$$

This calculation shows that the accumulated opacity is reduced to 0.84.

The key issue arises from the **dependency** between the probabilities of $\overline{AB}$, $\overline{AC}$, and $\overline{BC}$. In the above formulation, the term $(1.0 - 0.2) \cdot 0.8$ represents the probability that the ray misses $\overline{AB}$ and hits $\overline{BC}$. If the probabilities of $\overline{AB}$ and $\overline{BC}$ were independent, this formulation would be correct. However, they are dependent: in fact, the probability of $\overline{BC}$ equals $1.0 - \overline{AB}$ because both depend on the probability of C. Thus, the actual accumulated opacity should

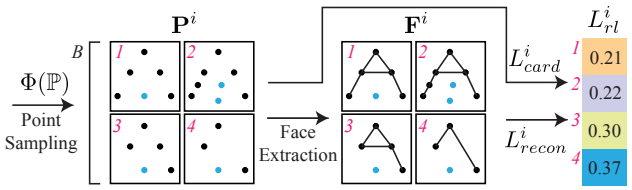


Figure 23. **Overview of *Reinforce-Ball* Algorithm.** Based on per-point existence probability ($\Phi(\mathbb{P})$), we sample points for B number of batches ($\mathbf{P}^i$). Here we use $B = 4$, and assume we are reconstructing shape "A". The points with $\psi = 1$ are rendered in black, while those with $\psi = 0$ are rendered in blue. Then, we identify existing faces in each batch ($\mathbf{F}^i$) based on Eq. (2). With $\mathbf{P}^i$ and $\mathbf{F}^i$, we compute loss for each batch. Note that the case (1, 2) are better than (3, 4), because they reconstruct the shape better (L^i_{recon}). Also, the case (1) is better than (2), because it has less number of points (L^i_{card}). To minimize the expected loss ($\mathbb{E}[L_{rl}]$), we should maximize the probability to sample the case 1. We optimize $\Phi(\mathbb{P})$ to do that.

be:

$$0.2 + (1.0 - 0.2) \cdot 1.0 = 1.0. \tag{17}$$

However, the alpha blending technique used here does not account for such dependencies, leading to a reduction in accumulated opacity. This reduction artificially increases the reconstruction loss. To minimize the loss, the optimizer increases the probability of C again, preventing convergence to the optimal case (3).

This dependency issue creates a local minimum that the previous formulation cannot overcome. This is why we propose the *Reinforce-Ball* algorithm.

10.2. Algorithm Overview

In the *Reinforce-Ball* algorithm, we define per-point existence probability and optimize it using stochastic optimization technique [48]. The overview of this algorithm is given in Fig. 23.

To elaborate, for a point $p \in \mathbb{P}$, let us denote the probability of it as $\phi(p) \in [0, 1]$, and concatenation of them as $\Phi(\mathbb{P})$. Then, assuming we sample points independently, we can sample a set of points $\mathbf{P}$ from $\Phi(\mathbb{P})$ and compute its probability as follows:

$$P(\mathbf{P}|\Phi(\mathbb{P})) = \Pi_{p\in\mathbf{P}}\phi(p) \cdot \Pi_{p\in\mathbb{P}-\mathbf{P}}(1 - \phi(p)). \tag{18}$$

Now, we sample points for B batches, and denote the sample points for i-th batch as $\mathbf{P}^i$. Based on $\mathbf{P}^i$ and tessellation function in Eq. (2), we can find out which faces exist for the i-th batch. Importantly, this process does not require evaluating all possible global face combinations; instead, it focuses only on local combinations, leveraging the *minimum-ball* condition in the tessellation function. We write these faces as $\mathbf{F}^i$, and use them for computing reconstruction loss for i-th batch (L^i_{recon}). We also compute "cardinality" loss for i-th batch (L^i_{card}), which is just the number of sampled points ($|\mathbf{P}^i|$). Then, we can compute the loss L^i_{rl} as

$$L^i_{rl} = L^i_{recon} + \epsilon_{card} \cdot L^i_{card}, \tag{19}$$

where ϵ_{card} is a small tunable hyperparameter to adjust the weight of the cardinality loss. If we write the final loss for a set of sampled points $\mathbf{P}$ as $L_{rl}(\mathbf{P})$, we aim at minimizing the expected loss:

$$\mathbb{E}_{\mathbf{P}\sim\Phi(\mathbb{P})}L_{rl}(\mathbf{P}) = \sum P(\mathbf{P}|\Phi(\mathbb{P})) \cdot L_{rl}(\mathbf{P}). \tag{20}$$

Algorithm 3 `Reinforce-Ball`

1: $n_0, n_1 \leftarrow$ Number of epochs and optimization steps
2: $B \leftarrow$ Number of batch samples
3: $\Phi \leftarrow$ Per-point probabilities, initialized to 0.99
4: $i \leftarrow 0$
5: **while** $i < n_0$ **do**
6: $\quad\mathbb{F} \leftarrow$ `Update-Query-Faces` $(\mathbb{P}, \Psi)$
7: $\quad B_{\mathbb{F}} \leftarrow$ `Compute-Minimum-Ball` $(\mathbb{P}, \mathbb{F})$
8: $\quad j \leftarrow 0$
9: $\quad$**while** $j < n_1$ **do**
10: $\quad\quad$*(k = 1, ..., B)*
11: $\quad\quad\mathbf{P}^k \leftarrow$ `Sample-Points` $(\mathbb{P}, \Phi)$
12: $\quad\quad\mathbf{F}^k \leftarrow$ `Get-Exist-Faces` $(\mathbf{P}^k, \mathbb{F}, B_{\mathbb{F}})$
13: $\quad\quad L^k_{rl} \leftarrow$ `Compute-Loss` $(\mathbb{P}, \mathbf{P}^k, \mathbf{F}^k)$
14: $\quad\quad\frac{\partial\mathbb{E}[L_{rl}]}{\partial\Phi} \leftarrow$ `Estimate-Gradient` $(\Phi, \mathbf{P}^k, L^k_{rl})$
15: $\quad\quad\Phi \leftarrow$ `Update-Gradient` $(\Phi, \frac{\partial\mathbb{E}[L_{rl}]}{\partial\Phi})$
16: $\quad$**end while**
17: $\quad\mathbb{P}, \Psi \leftarrow$ `Get-Remaining-Points` $(\mathbb{P}, \Psi, \Phi)$
18: **end while**

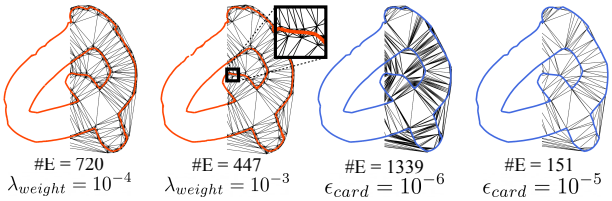
10.3. Formal Definition

In Algorithm 3, we formally describe the *Reinforce-Ball* algorithm in detail:

- **Line 1:** In the *Reinforce-Ball* algorithm, we optimize per-point probabilities for n_0 epochs, with each epoch consisting of n_1 optimization steps. In our experiments, we set $n_0 = 10$ and $n_1 = 2000$.
- **Line 2:** We define the number of batches used during optimization as B. Increasing B improves the stability of the gradient computation but also increases computational cost. In our experiments, we set $B = 1024$.
- **Line 3:** Initialize the per-point probability of every point to 0.99, as all points are assumed to exist with high probability before optimization. The probabilities are not set to 1.0 to avoid every sampled batch (Line 11) including all points, which would prevent optimization from progressing.
- **Lines 4-5:** Perform multiple epochs of optimization.
- **Line 6:** Gather the possibly existing faces ($\mathbb{F}$) based on the current point configuration and their real values. This

Method (hyperparameter)	CD($\times 10^{-6}$)↓	# Verts.	# Edges.	Time (sec)
DMesh [42] (0)	1.97	2506	2245	30.39
DMesh (10^{-4})	2.68	666	693	153.10
DMesh (10^{-3})	12.48	456	488	152.37
DMesh++ (0)	1.82	2862	2793	11.33
DMesh++ (10^{-6})	1.86	1386	1394	278.88
DMesh++ (10^{-5})	2.77	149	152	200.05

Table 6. **Quantitative ablation studies on Reinforce-Ball algorithm.** As we increase ϵ_{card} (in parenthesis) for DMesh++, we can significantly reduce the mesh complexity without losing geometric details, while DMesh cannot do the same with λ_{weight}.



(a) DMesh (b) DMesh++

Figure 24. **Qualitative ablation studies on Reinforce-Ball algorithm (for letter 'Q').** We render "imaginary" (black) part and "real part" (red, blue) together.

function is the same as the one used in the Point Optimization step (Appendix 8.2.2).

- **Line 7:** Compute the minimum bounding ball $B_{\mathbb{F}}$ for the gathered query faces.
- **Lines 8-9:** Perform the optimization steps within the current epoch.
- **Line 10:** Consider B batches, each containing a different point configuration based on the sampled points.
- **Line 11:** For each batch, sample points from $\mathbb{P}$ based on their probabilities Φ. Each point is sampled independently, and the probability of sampling a specific batch is computed as shown in Eq. (18). The sampled points in the k-th batch are denoted as $\mathbf{P}^k$.
- **Line 12:** For each batch, determine the existing faces in $\mathbb{F}$ based on the sampled points. Specifically, a face F exists if all its points are included in the sampled points and its B_F satisfies the *Minimum-Ball* condition. The existing faces in the k-th batch are denoted as $\mathbf{F}^k$.
- **Line 13:** For each batch, compute the loss as the sum of the reconstruction loss (L_{recon}) and the cardinality loss (L_{card}), as discussed in Appendix 10.2.
- **Line 14:** Estimate the gradient of the expected loss ($\mathbb{E}[L_{rl}]$) with respect to the per-point probabilities Φ using the log-derivative trick [48]:

$$\nabla_{\Phi}\mathbb{E}_{\mathbf{P}\sim\Phi}[L_{rl}] \approx \frac{1}{B}\sum_{i=1}^{B}\nabla_{\Phi}\log P(\mathbf{P^i}|\Phi)\cdot L^i_{rl}. \quad (21)$$

  To reduce the variance of the gradient, we normalize L_{rl} across the batch before the computation [11].
- **Line 15:** Update Φ using the estimated gradients.
- **Line 17:** After completing an epoch, discard points whose probability is below a specified threshold. In our experiments, we set the threshold to 0.5. The remaining points are used for the next epoch. As points are removed, the query faces updated in Line 6 for the next epoch will span a larger area than in the previous epoch.

10.4. Experimental Results

Using the Reinforce-Ball algorithm, we can reconstruct efficient 2D meshes from point clouds that adapt to local geometry. As described in Sec. 5.2.1, we conducted 2D point cloud reconstruction experiments on the font dataset.

In Tab. 6, we present quantitative ablation studies on the *Reinforce-Ball* algorithm. Increasing the tunable hyperparameter ϵ_{card} (Appendix 10.2), which controls regularization strength, leads to a rapid reduction in vertices and edges. For instance, with $\epsilon_{card} = 10^{-5}$, edges decrease by nearly *94%* with minimal impact on reconstruction quality. DMesh [42] also offers a tunable parameter, λ_{weight}, for weight regularization to reduce mesh complexity. However, while edge reduction occurs, DMesh's reconstruction quality degrades more quickly. At $\epsilon_{card} = 10^{-5}$, our method achieves a similar CD loss to DMesh with $\lambda_{weight} = 10^{-4}$ but uses about *78%* fewer edges. This advantage is also evident in Fig. 24, where our *Reinforce-Ball* algorithm removes redundant edges effectively and adapts the mesh to local geometry. In contrast, DMesh's edge removal disregards local geometry, resulting in loss of detail.

Likewise, we successfully highlighted the limitation of DMesh's weight regularization and demonstrated that the *Reinforce-Ball* algorithm can eliminate redundant mesh faces within the DMesh++ framework without sacrificing geometric details. However, since the method is not yet easily extensible to 3D and incurs high computational costs, we include these results in the Appendix.