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

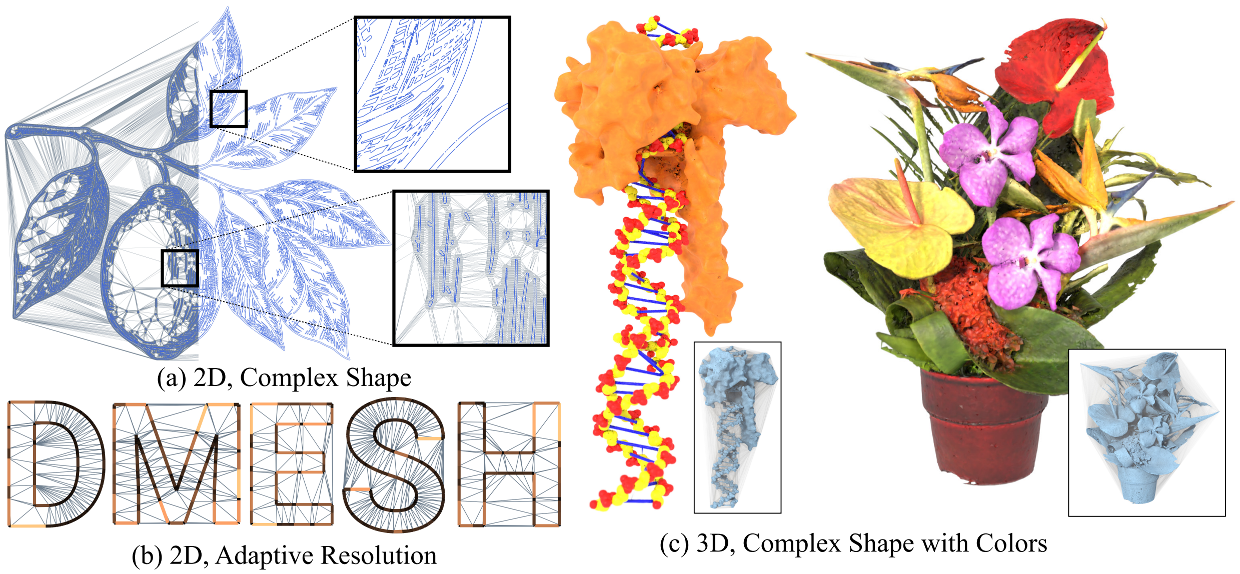

(a) 2D, Complex Shape
(b) 2D, Adaptive Resolution
(c) 3D, Complex Shape with Colors

Figure 1. **DMesh++ for complex 2D and 3D shapes.** (a) DMesh++ can reconstruct complex 2D drawings from sample points extracted from them. (b) In 2D, DMesh++ can produce efficient yet accurate meshes that adapt to local geometry. (c) DMesh++ can also reconstruct complex 3D shapes with colors from point clouds and 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). In (b), the tone of the color represents the edge length.

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 in 2D and 3D that addresses this challenge and efficiently handles meshes with intricate structures. Additionally, we present an algorithm that adapts the mesh resolution to local geometry in 2D for efficient representation. We demonstrate the effectiveness of our approach on 2D point cloud and 3D multi-view reconstruction tasks. Visit our project page (https://sonsang.github.io/dmesh2-project) for source code and supplementary material.

* This work was done during internship at Adobe Research.

** The paper was last modified on Dec. 20th, 2024.

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 fully differentiable shape representations. To address this, recent data-driven efforts have attempted to predict mesh connectivity using Transformer-based models [5, 6, 11, 35, 36]. However, these methods face inherent challenges with robustness to outlier meshes, potential self-intersections, and high computational costs.

On another route, Son et al. [37] introduced a new form 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. This approach guarantees mesh without self-intersections, and is free from outliers, as it is not data-driven. Therefore, this probabilistic approach opens up a new venue to adopt meshes in a machine learning pipeline, such as generative models [41, 45]. However, it suffers from excessive computational cost when the number of points increases (Fig. 6), 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. This algorithm uses minimum circumscribing ball of a face to compute its probability. 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 under practical scenarios (Sec. 3.2 and Fig. 6).

The direct application of DMesh++ is a reconstruction task. Using *Minimum-Ball* algorithm, we can reconstruct complex 2D and 3D meshes (Figs. 1, 7 and 11) efficiently within a reasonable time-frame. At the same time, as the reconstructed mesh usually has lots of redundant faces, we propose *Reinforce-Ball* algorithm to remove them, and produce an efficient mesh for 2D cases (Sec. 4.2). It is designed to optimize the probability of a point's existence, and minimize the number of points while keeping the geometric accuracy (Tab. 1 and Figs. 1 and 8). Additionally, we propose several other novelties in reconstruction process (Secs. 4.3 and 4.4) for better reconstruction.

To summarize, our contributions are the following:

- We present DMesh++, an enhanced version of DMesh [37] that addresses its computational bottlenecks with *Minimum-Ball* algorithm.
- To eliminate redundant faces and produce an efficient mesh that adapts to local geometry in 2D reconstruction, we introduce *Reinforce-Ball* algorithm.
- We propose additional mesh operations to use in the reconstruction pipeline. We also present an improved differentiable renderer for 3D multi-view reconstruction.
- Utilizing these components, we reconstruct high-quality meshes of complex shapes in both 2D and 3D, demonstrating our approach's effectiveness in 2D point cloud reconstruction and 3D multi-view reconstruction tasks.

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. We briefly introduce them below.

Using Alternative Differentiable Shape Representations. Rather than handling mesh directly, some prior work extract mesh from alternative differentiable shape representations. Neural implicit representations, like signed or unsigned distance fields [18, 20, 28, 30, 38–40, 43, 44], encode distance fields in neural networks, and use iso-surface extraction methods [10, 13, 21] to generate the mesh. Another method encodes distance directly into spatial points and applies differentiable iso-surface extraction [16, 19, 22, 24, 33, 34, 40]. While often more efficient, these methods typically cannot handle open surfaces; though [19] does, it cannot represent non-orientable geometries. Gaussian Splatting [14] 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 (almost) fixed mesh connectivity [4, 15, 17, 27, 29, 47]. Recently, data-driven approaches have aimed to overcome these limitations by training generative models [5, 6, 11, 35, 36] that predict vertex connectivity from point clouds. Specifically, SpaceMesh [35] ensures combinatorial manifold mesh generation. However, these models still struggle with outliers and are susceptible to self-intersections.

Designing A Differentiable Form of Mesh. Son et al. [37] recently introduced DMesh, a differentiable mesh formulation using a probabilistic approach. DMesh augments each point with two continuous values, along with its position, and applies a "tessellation" function in Eq. (1) to deterministically generate a mesh from a point set. This method adapts to various geometric topologies and avoids self-intersections. With DMesh, optimizing or inferring only point-wise features is sufficient to generate the mesh, simplifying the application of losses on points or faces.

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 [12]. For $N = 100K$ in 3D, the runtime can reach up to 800 milliseconds (Fig. 6), limiting DMesh's applicability for complex shapes requiring finer detail. In this work, we eliminate WDT, proposing 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 one of our primary algorithms.

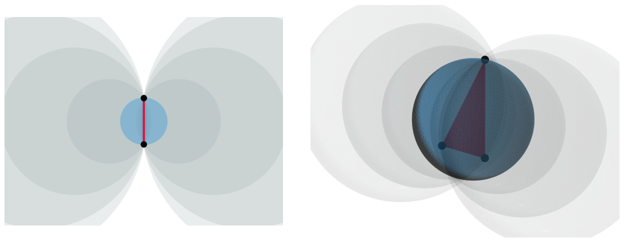


Figure 2. **Bounding balls of a face F (red) in 2D (left) and 3D (right).** The minimum bounding ball (B_F) is rendered in blue, while the others are rendered in gray.

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 [37] 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:

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

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, $(x_1, ..., x_d, \psi)$. 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 it. However, the tessellation function is computationally expensive, as we need to compute WDT to define $\mathbb{F}_{wdt}$. In designing our *Minimum-Ball*, we desire to remove this necessity for acceleration, while inheriting the nice property about self-intersection. In the following, we show that *Minimum-Ball* satisfies these requirements.

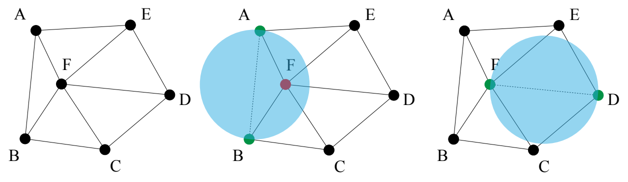


Figure 3. **Minimum-Ball condition in 2D.** In the left, 2D Delaunay Triangulation (DT) of 6 points is given. In middle and right figure, we render B_F for two faces ($\overline{AB}$, $\overline{DF}$) in blue.

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. 2). Then, we define $\mathbb{F}_{min}$ as a set of faces whose minimum bounding ball does not contain any other point in $\mathbb{P}$.

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. 3, 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 $\mathbb{P}$, which we denote as $\mathbb{F}_{dt}$.

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 $\mathbb{P}$ [7]. 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 $\mathbb{F}_{min}$. 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. 3).

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 $B^c_F \in \mathbb{R}^d$ and B^r_F. We can compute these values in a differentiable way (Appendix 7.2). Then, we can compute the maximum signed distance between B_F and $\mathbb{P}$ as follows:

$$d(B_F, \mathbb{P}) = \max_{p \in \mathbb{P}-F} B^r_F - ||p - B^c_F|| \quad (3)$$
$$= B^r_F - \min_{p \in \mathbb{P}-F} ||p - B^c_F||. \quad (4)$$

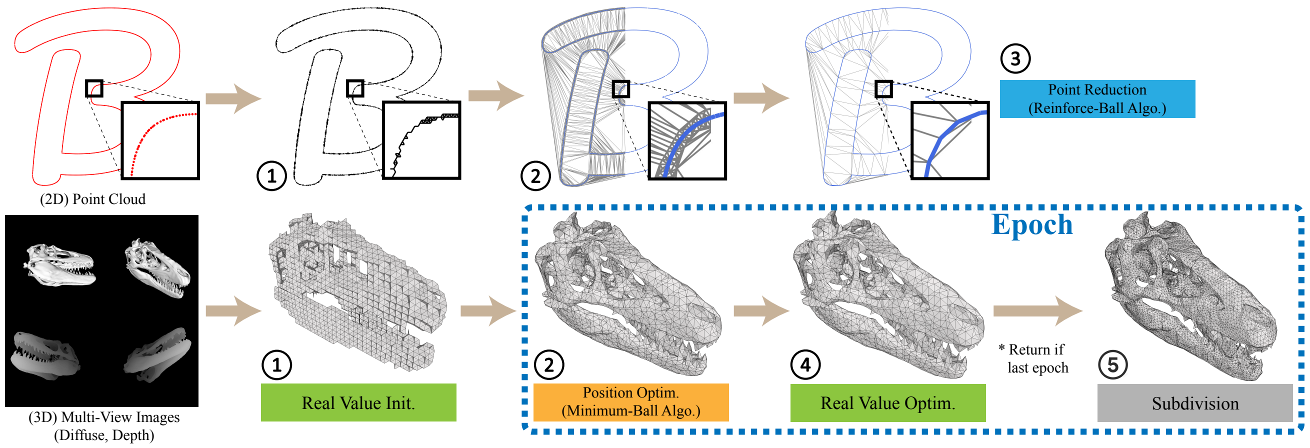

(2D) Point Cloud
①
②
③
Point Reduction (Reinforce-Ball Algo.)
(3D) Multi-View Images (Diffuse, Depth)
①
Real Value Init.
Epoch
②
Position Optim. (Minimum-Ball Algo.)
④
Real Value Optim.
* Return if last epoch
⑤
Subdivision

Figure 4. **Reconstruction pipeline for 2D point cloud (up) and 3D multi-view images (down).** Color represents per-point feature that is optimized: position, real (ψ), and probability (ϕ).

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.

$$F \in \mathbb{F}_{min} \Leftrightarrow d(B_F, \mathbb{P}) > 0. \tag{5}$$

Then, it is straightforward to define Λ_{min} with sigmoid function as

$$\Lambda_{min}(F) = \sigma(d(B_F, \mathbb{P}) \cdot \alpha_{min}), \tag{6}$$

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[1], 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 $O(\log |\mathbb{P}|)$[2]. This allows our tessellation function to run up to 32 times faster in 3D than DMesh [37] (Fig. 6). 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

In this section, we discuss how to reconstruct DMesh++ in 2D and 3D from point clouds or multi-view images, as reconstruction is an immediate application of DMesh++. We introduce the reconstruction pipeline and novel algorithms to enhance mesh quality, including our second main algorithm, *Reinforce-Ball*. We also provide an overview of relevant loss functions.

[1]We used implementation of PyTorch3D [32].

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

4.1. Pipeline Overview

The goal of reconstruction is to optimize point-wise features, so that the resulting mesh aligns with the input observation – either a point cloud (2D) or multi-view images (3D) as shown in Figure 4. For implementation details about each step, please refer Appendix 8.

For 2D point cloud, we initialize points on a fixed triangular grid and set the "real" values based on their overlap with the point cloud (Step 1). We then fix the "real" values and optimize only the point positions (Step 2), using the *Minimum-Ball* algorithm to compute face existence probabilities (Sec. 3.2). To achieve an efficient yet geometrically accurate mesh, we use a novel *Reinforce-Ball* algorithm (Sec. 4.2) in Step 3.

For multi-view images, we initialize the 3D points on a tetrahedral grid and set the "real" values based on their projections onto the images (Step 1). As before, we fix the "real" values initially, optimizing only the point positions based on *Minimum-Ball* (Step 2), then proceed to optimize the "real" values again with fixed point positions (Step 3). Then, if desired, we remove non-manifoldness at this step. To achieve higher resolution, we apply a face subdivision scheme between epochs (Step 4). We propose algorithms for these mesh operations later (Sec. 4.3).

4.2. Reinforce-Ball Algorithm

Although the *Minimum-Ball* approach enables high-resolution mesh handling, it often results in overly dense meshes with redundant faces after reconstruction (Fig. 4). This occurs because *Minimum-Ball* lacks mesh complexity regularization. To address this in 2D cases, we propose

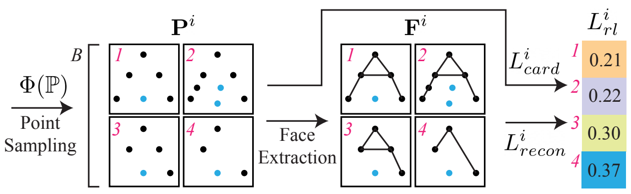
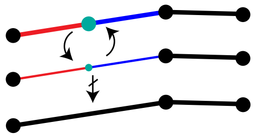


Figure 5. **Overview of *Reinforce-Ball* Algorithm.** Based on per-point existence probability ($\Phi(\mathbb{P})$), we sample points for B number of batches ($\mathbf{P}^i$). Here we use $B = 4$, and assume we are reconstructing shape "A". The points with $\psi = 1$ are rendered in black, while those 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.

Reinforce-Ball algorithm, which reduces unnecessary faces while preserving geometric detail.



Previously, DMesh [37] used "weight regularization" to simplify meshes by increasing regularization strength. However, this approach struggled with adaptive resolution due to local minima (Fig. 8). For example, when reconstructing the left line segment in inset, only the end points (black) are necessary, while the intermediate point (green) and colored edges are redundant. The bottom configuration is ideal, so if optimization starts from the top configuration, weight regularization reduces the redundant point's weight, as shown in the middle configuration. However, this also lowers the probabilities of redundant edges, increasing reconstruction loss and thus restoring the redundant point's weight, preventing convergence to the optimal configuration (Appendix 9.2). This issue arises from the combinatorial nature of the problem.

To solve this problem, our *Reinforce-Ball* algorithm defines per-point existence probability and optimize it using stochastic optimization technique [42]. The overview of this algorithm and its formal definition is given in Fig. 5 and Appendix 9.1, respectively.

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:

$$P(\mathbf{P}|\Phi(\mathbb{P})) = \Pi_{p\in\mathbf{P}}\phi(p) \cdot \Pi_{p\in\mathbb{P}-\mathbf{P}}(1 - \phi(p)). \tag{7}$$

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

$$L^i_{rl} = L^i_{recon} + \epsilon_{card} \cdot L^i_{card}, \tag{8}$$

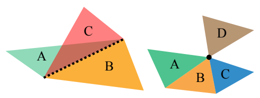
where ϵ_{card} is a small tunable hyperparameter to adjust the weight of the cardinality loss. We report ablation on it in Tab. 1 and Fig. 8. If we write the final loss for a set of sampled points $\mathbf{P}$ as $L_{rl}(\mathbf{P})$, we aim at minimizing the expected loss:

$$\mathbb{E}_{\mathbf{P}\sim\Phi(\mathbb{P})}L_{rl}(\mathbf{P}) = \sum P(\mathbf{P}|\Phi(\mathbb{P})) \cdot L_{rl}(\mathbf{P}). \tag{9}$$

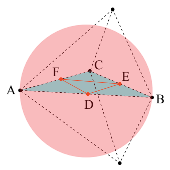
We can estimate the gradient of this expected loss using log-derivative trick [25]. Then, we can update $\Phi(\mathbb{P})$ using this gradient, and remove unnecessary points while preserving the original geometry (Figs. 1, 4 and 8). Notably, this algorithm is particularly effective in 2D but less so in 3D, as explained in Appendix 9.3.

4.3. Mesh Operations

Here we propose two mesh operations that we use in the 3D reconstruction pipeline, which are easily applicable to 2D.



Non-Manifoldness Removal Mesh defined by our tessellation function (Eq. (2)) is not free from non-manifold edges or vertices in combinatorial sense. We remove non-manifold edges and vertices in Step 4. Non-manifold edges (inset, left) are edges that are adjacent to more than 2 faces, and non-manifold vertices (inset, right) are vertices whose adjacent faces do not form single fan. We detect such cases, and remove faces that are adjacent to such non-manifoldness until all of such cases are resolved, based on their contribution to the reconstruction loss (*e.g.* number of rendered pixels per face).



Mesh Subdivision For mesh subdivision, we add 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 (Appendix 10).

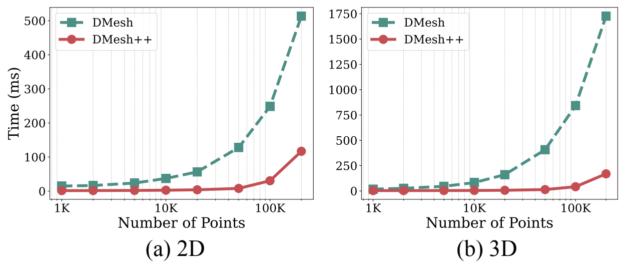


Figure 6. **Comparison of tessellation speed.** Our method based on minimum-ball algorithm computes face probabilities up to 16 times (2D) and 32 times (3D) faster than DMesh [37].

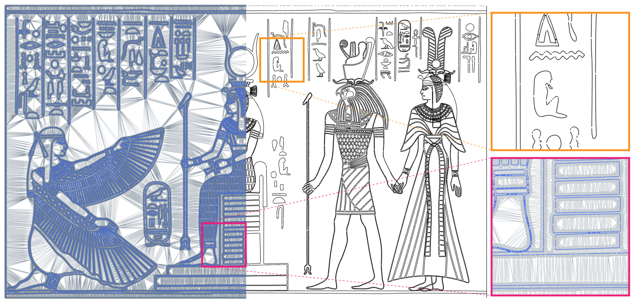
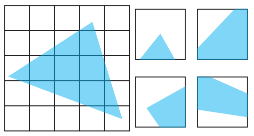


Figure 7. **Quality of DMesh++ reconstructed from 2D point cloud sampled from Egyptian painting.** In left part, "imaginary" part (gray) and "real" part (blue) are rendered together.

4.4. Loss Functions



For 2D point cloud reconstruction, we use Chamfer Distance (CD) as the reconstruction loss. For 3D multi-view reconstruction task, we use rendering loss. Note that we need to formulate such loss functions to consider face probabilities (Λ, Sec. 3.1) in Step 1, 2, and 4. We follow formulations of DMesh [37] – for CD, they propose expected CD, and for rendering, they interpret face probability as its opacity in their own differentiable renderer. However, their differentiable renderer lacks visibility gradients, and has to rely on another rendering process for them. To amend this issue, we improved their renderer by implementing (face) anti-aliasing in CUDA. As shown in the inset, we compute the intersecting area between each pixel and triangle to compute the face opacity at the pixel (Appendix 11). Finally, we use triangle quality regularization [37] to improve the mesh quality.

5. Experiments

This section presents our experimental results. First, we evaluate how *Minimum-Ball* accelerates the tessellation function in 2D and 3D. Next, we demonstrate the *Reinforce-Ball* algorithm's effectiveness in producing efficient 2D meshes with adaptive resolution. Finally, we showcase a practical application in 3D multi-view reconstruction, illustrating that our method is suitable for (differentiable) downstream tasks involving complex shapes. Our main algorithms are implemented in PyTorch [31], with a differentiable 3D renderer in CUDA [26]. All experiments were conducted on a system with an AMD EPYC 7R32 CPU and Nvidia A10 GPU.

Method (hyperparameter)	CD($\times 10^{-6}$)↓	# Verts.	# Edges.	Time (sec)
DMesh [37] (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 1. **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}.

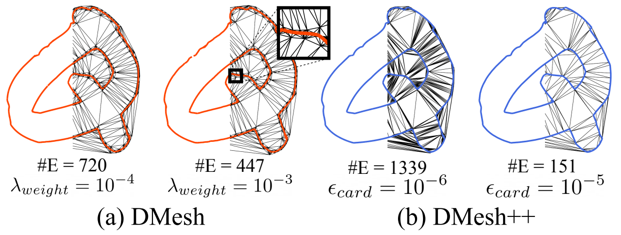

#E = 720
$\lambda_{weight} = 10^{-4}$
#E = 447
$\lambda_{weight} = 10^{-3}$
#E = 1339
$\epsilon_{card} = 10^{-6}$
#E = 151
$\epsilon_{card} = 10^{-5}$
(a) DMesh
(b) DMesh++

Figure 8. **Qualitative ablation studies on Reinforce-Ball algorithm (for letter 'Q').** We render "imaginary" (black) part and "real part" (red, blue) together.

5.1. Tessellation Speed

We compare the tessellation function speed between DMesh [37] (Eq. (1)) and our DMesh++ (Eq. (2)). For both 2D and 3D cases, we randomly generate N points in 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 N (ranging from $1K$ to $200K$ to reflect practical scenarios), we ran five trials and averaged the computational speeds.

In Fig. 6, we compare the computational costs of DMesh and DMesh++. For DMesh, costs increase linearly with point count in both 2D and 3D, due to the sequential WDT algorithm. In contrast, DMesh++ shows sub-linear scaling up to 50K points, benefiting from GPU parallelization (Sec. 3.2). Beyond this, costs rise more sharply due to GPU thread limitations. Nevertheless, DMesh++ handled 200K points in $117ms$ for 2D and $168ms$ for 3D. This demonstrates that the *Minimum-Ball* algorithm significantly accelerates tessellation, enabling efficient handling of complex shapes (Figs. 1 and 7) that DMesh cannot handle. See Appendix 13.1 for more results on complex 2D drawings.

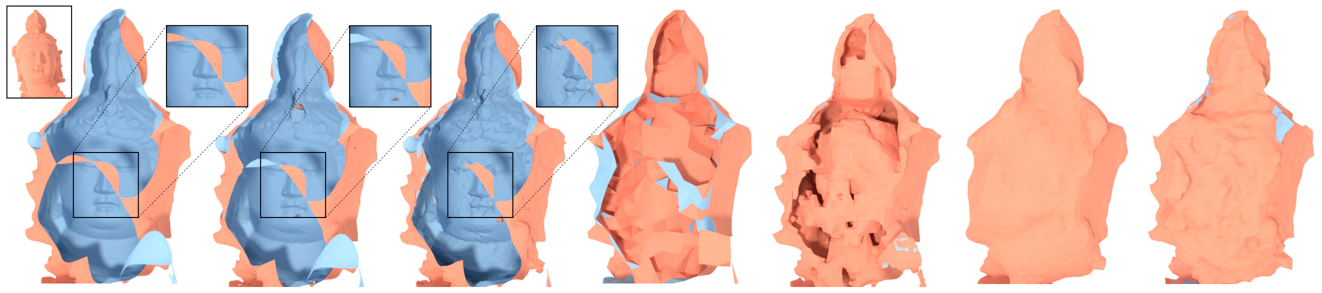

(a) Ground Truth (b) DMesh++ (c) GShell (d) DMesh (e) FlexiCubes (f) DMTet (g) Remeshing

Figure 9. **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. DMesh++ captures geometric details better than the other methods, while GShell [19] suffers from intricate structure of the shape, and DMesh [37] produces false inner structures. The other approaches cannot represent open surfaces by nature.

	Geometric Accuacy					Mesh Quality				Statistics		
Method	CD($\times 10^{-3}$)↓	F1↑	NC↑	ECD↓	EF1↑	AR↓	SI↓	NME↓	NMV↓	# Verts.	# Faces.	Time (sec)
Remeshing [29]	1.208	0.427	0.941	0.178	0.025	1.498	0.264	0	0	11895	23785	25
DMTet [33]	0.923	0.251	0.926	0.228	0.017	6.466	0.001	0	0	18145	36315	201
FlexiCubes [34]	1.032	0.355	0.925	0.154	0.025	2.065	0.014	0	0.003	12570	24745	88
GShell [19]	0.542	0.402	0.939	0.150	0.077	2.784	0.063	0	0.002	11920	23137	208
DMesh [37]	0.494	0.393	0.937	0.116	0.060	1.799	0	0.032	0.001	2396	4863	751
DMesh++(Ours)	0.127	0.460	0.964	0.133	0.062	1.603	0	0	0	10021	19300	204

Table 2. **Quantitative comparison of 3D multi-view reconstruction results.** We evaluate the reconstruction results with 5 metrics related to geometric accuracy and 4 metrics related to mesh quality. For geometric accuracy, we use Chamfer Distance (CD), F1 score (F1), Normal Consistency (NC), Edge Chamfer Distance (ECD), and Edge F1 score (EF1). For mesh quality, we use face Aspect Ratio (AR), Self-Intersection face Ratio (SI), Non-Manifold Edge ratio (NME), and Non-Manifold Vertex ratio (NMV). Finally, we additionally report several statistics: number of vertices and faces, and computation time. We highlighted the best results and the second best results for the evaluation metrics. For every criteria, DMesh++ achieves the best, or at least comparable results.

5.2. Adaptive Resolution

We now demonstrate the impact of the *Reinforce-Ball* algorithm on 2D point cloud reconstruction for a font dataset. This experiment uses vector graphics of 26 uppercase letters from four different font styles. For each letter, we sampled a dense point cloud, which helps prevent "holes" that can occur with insufficient point density (Appendix 12.1).

In Tab. 1, we present quantitative ablation studies on the *Reinforce-Ball* algorithm. Increasing the tunable hyperparameter ϵ_{card} (Sec. 4.2), 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 [37] 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. 8, 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. The main limitation of *Reinforce-Ball* is its computational cost; while memory-efficient, it converges slowly. Accelerating this algorithm would be a valuable direction for future work.

5.3. 3D Multi-View Reconstruction

In this task, we reconstruct a mesh from multi-view images of a target object. We render (512×512) diffuse and depth maps of the ground truth object from 64 viewpoints (Fig. 4). We selected 10 closed and 10 open surfaces from Thingi10K dataset [46] for this experiment. We chose objects with minimal self-occlusions, as we need dense observations of an object to fully reconstruct it (Appendix 12.2). For comparison, we tested five other mesh reconstruction algorithms: Remeshing [2], DMTet [33], FlexiCubes [34], GShell [19], and DMesh [37], each with settings optimized for best quality, including post-processing steps such as visibility tests to remove false internal structures. We also aimed to produce meshes of similar complexity. See Appendix 12.2 for details.

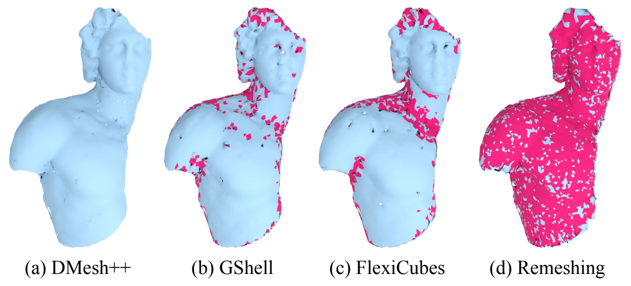


Figure 10. **Self-intersection of the reconstructed mesh.** The self-intersected faces of the mesh are rendered in red.

In Tab. 2 and Figs. 9 and 10, we present quantitative and qualitative comparisons of the reconstruction results. For quantitative analysis, we assess geometrical accuracy and mesh quality using five and four metrics, respectively, and also report statistics on mesh complexity and computation time. The values represent averages across the entire dataset; in Appendix 13.2, we separate results for closed and open surfaces.

In Tab. 2, we observe that Remeshing, DMTet, and FlexiCubes have high CD errors, largely because they cannot represent open surfaces (Fig. 9). This limitation also explains why Remeshing and FlexiCubes are faster than other methods. Although Remeshing achieved the best AR and avoided non-manifoldness, it generated numerous self-intersections, particularly on open surfaces (Fig. 10).

GShell improves CD loss by representing open surfaces through sub-surface extraction from closed templates. However, it struggles with complex open surfaces, as estimating the closed templates is challenging at the first place (Fig. 9). It also suffers from self-intersections (Fig. 10) and suboptimal AR. Compared to that, DMesh produced self-intersection free mesh with better AR and CD, as it can represent open surfaces. Also, it produced much simpler mesh than the other methods. However, it sometimes produced false inner structure due to occlusion (Fig. 9), and was not free from non-manifoldness. Its largest drawback was in computational cost, limiting its utility for fine-grained reconstructions.

DMesh++ addresses this issue with the *Minimum-Ball* algorithm (Sec. 3.2) and nearest neighbor caching (Appendix 7.4). It achieves the best or comparable results across all metrics, with computational costs similar to other methods. Importantly, DMesh++ guarantees meshes free of self-intersections (Fig. 10) and non-manifoldness, offering geometrically accurate and high-quality reconstructions at efficient computational costs.

Colors. For 3D multi-view reconstruction, we jointly optimize per-point colors to recover a textured object or small scene. For a face $F = \{p_1, p_2, p_3\}$, the color of a point p on F is determined by interpolating the colors of p_1, p_2, and p_3 using barycentric coordinates. In our experiments, we used a similar setup as before, substituting diffuse maps with high-resolution colored renderings (Appendix 12.2). We reconstructed textured meshes from several objects and small scenes in Objaverse [8] (Figs. 1 and 11), demonstrating our method's ability to handle textured meshes and potential for real-world image-based reconstruction. More in Appendix 13.3.

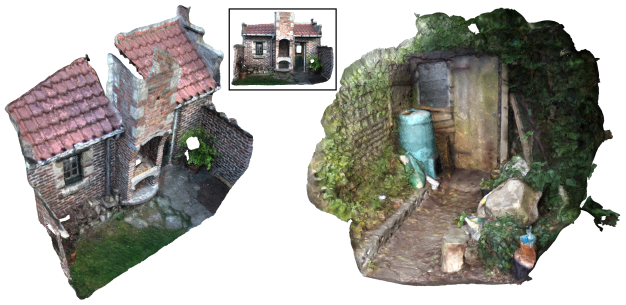


Figure 11. **3D scene reconstructions from multi-view images.** For the left scene, we render the front view at the top right corner.

6. Discussion

We presented DMesh++, a probabilistic approach for efficiently managing mesh connectivity. While primarily demonstrated for reconstruction task on synthetic dataset, DMesh++ can be applied across a range of applications.

First, we can extend the current reconstruction pipeline to create large scenes from real-world images. Although other representations, like NeRF [23] and Gaussian Splatting [14], yield high-quality reconstruction results, they lack precise geometric information. DMesh++, however, directly produces meshes that can be readily used in downstream applications, such as physics simulations. We also envision DMesh++ being used to train generative models to understand mesh connectivity. Since DMesh++ accommodates diverse mesh topologies, generative models trained on it could produce meshes with varied topologies. For example, it could be applied to generate or infer complex structures in Biometrics like DNA (Fig. 1).

Limitations and Future Directions. DMesh++ has several limitations to address for broader utility. First, although DMesh++ offers a way to eliminate non-manifoldness in a combinatorial sense, this does not ensure a favorable mesh topology, such as watertight mesh. Removing faces to resolve non-manifoldness often creates small holes in the mesh surface, which are undesirable. Second, we only implement reconstruction of DMesh++ from input images or point clouds. Depending purely on inputs, issues like self-occlusions in multi-view reconstruction can lead to floating artifacts (Appendix 12.2). To handle this, future works can introduce topological constraints to DMesh++ and utilize data-driven priors to facilitate the reconstruction.

Acknowledgement This work was supported in part by Barry Mersky and Capital One E-Nnovate Endowed Professorships, National Science Foundation, U.S. Army Research Office, and Meta Graduate Fellowship.

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] Shek Ling Chan and Enrico O Purisima. A new tetrahedral tesselation scheme for isosurface generation. *Computers & Graphics*, 22(1):83–90, 1998. 3

[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] Siu-Wing Cheng, Tamal Krishna Dey, Jonathan Shewchuk, and Sartaj Sahni. *Delaunay mesh generation*. CRC Press Boca Raton, 2013. 3

[8] 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. 8, 9, 10

[9] 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. 5

[10] 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

[11] Zekun Hao, David W Romero, Tsung-Yi Lin, and Ming-Yu Liu. Meshtron: High-fidelity, artist-like 3d mesh generation at scale. *arXiv preprint arXiv:2412.09548*, 2024. 1, 2

[12] Clément Jamin, Sylvain Pion, and Monique Teillaud. 3D triangulations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023. 2

[13] 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

[14] 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, 3

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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, 9, 10

[20] 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

[21] 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

[22] 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

[23] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021. 8

[24] 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

[25] Kevin P Murphy. *Probabilistic machine learning: an introduction*. MIT press, 2022. 5

[26] 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. 6

[27] 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

[28] 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

[29] Werner Palfinger. Continuous remeshing for inverse rendering. *Computer Animation and Virtual Worlds*, 33(5):e2101, 2022. 2, 7, 9, 10

[30] 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

[31] 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. 6

[32] 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

[33] 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, 9, 10

[34] 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, 9, 10

[35] 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

[36] 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

[37] 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, 9, 10

[38] 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

[39] 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.

[40] 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

[41] 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

[42] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992. 5

[43] 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

[44] 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

[45] 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

[46] Qingnan Zhou and Alec Jacobson. Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797*, 2016. 7, 8

[47] 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

In this work, we presented DMesh++, a novel approach to handle mesh in a differentiable manner.

As a core principle, we first introduced the *Minimum-Ball* algorithm (Sec. 3.2). This algorithm allows us to *increase* the mesh complexity in a computationally efficient manner, enabling the handling of complex shapes. Specifically, we demonstrated that this algorithm has a time complexity of $O(\log N)$, where N is the number of points. This efficiency is achieved because the *Minimum-Ball* algorithm considers only the **local** point configuration near a face to determine the face's existence. In contrast, DMesh [37] evaluates the **global** point configuration, resulting in a time complexity of $O(N)$, which is significantly more costly as N grows.

Under the *Minimum-Ball* condition, we select faces from the Delaunay Triangulation (DT) that satisfy this criterion. Among these, we identify faces whose vertices all have a real value (ψ) of 1. These faces constitute the "real" part and form the final mesh. The remaining faces in the DT, which do not satisfy this criterion, are labeled as the "imaginary" part and are excluded from the final mesh (Fig. 1).

Additionally, we presented the *Reinforce-Ball* algorithm (Sec. 4.2) for 2D reconstruction tasks. This algorithm allows us to *decrease* the mesh complexity, producing a mesh that adapts to local geometry. Like the *Minimum-Ball* algorithm, the *Reinforce-Ball* algorithm evaluates only the **local** neighborhood of a face to determine its existence.

To summarize, these two algorithms enable us to achieve both **computational efficiency** and **mesh efficiency**. In this appendix, we provide detailed explanations and further discussions on various aspects of DMesh++.

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. Below, we provide a line-by-line explanation of the algorithm:

- **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.

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:

$$B^c_F\big|_{d=2} = \frac{1}{2}(p_1 + p_2). \quad (10)$$

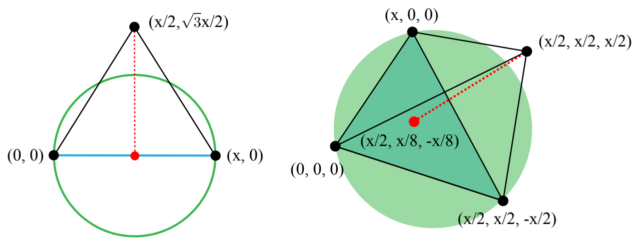


Figure 12. **Common signed distance for a 2D (left) and 3D (right) face in (initial) regular grid.** We compute the signed distance by subtracting the radius of the minimum bounding ball from the length of the red line. The red dot represents the center of the minimum bounding ball.

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

$$B_F^c\big|_{d=3} = p_1 + \frac{||d_2||^2(d_1 \times d_2) \times d_1 + ||d_1||^2(d_2 \times d_1) \times d_2}{2||d_1 \times d_2||^2}, \tag{11}$$

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_F^c 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_F^c, we calculate the radius B_F^r as the distance between B_F^c 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 than 0.5. To minimize such mismatches, we set α_{min} based on the density of the grid from which optimization begins.

As illustrated in Figure 4, the reconstruction process starts from a fixed triangular (2D) or tetrahedral (3D) grid (Fig. 13). At the initial state, every face in the grid satisfies the *Minimum-Ball* condition (Appendix 8.2). 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 $d = 2$:

$$d_{common} = \frac{\sqrt{3} - 1}{2} x. \tag{12}$$

For $d = 3$:

$$d_{common} = \frac{\sqrt{34} - 3\sqrt{2}}{8} x. \tag{13}$$

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

In Fig. 12, 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_{min}^1 represents the value in the first epoch, the value for the i-th epoch is given by:

$$\alpha_{min}^i = \frac{\alpha_{min}^1}{2^{i-1}}. \tag{14}$$

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_F^c 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 ($B_{\mathbb{F}}^c$). Then, we identify the K-nearest neighbors of $B_{\mathbb{F}}^c$ 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_F^c to the cached K-nearest neighbors in $\mathbb{P}$ and select the nearest neighbor from the cache to compute the signed distance $d(B_F, \mathbb{P})$. This mechanism is described in detail in Algorithm 2 and Appendix 8.3 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 Pipeline

In this section, we provide implementation details for each step in the optimization pipeline (Fig. 4). Before delving into these details, we revisit the reconstruction loss used in our framework.

8.1. Reconstruction Loss (L_{recon})

Point Cloud When ground truth point clouds are provided, we utilize the expected Chamfer Distance (CD) proposed by [37]. 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

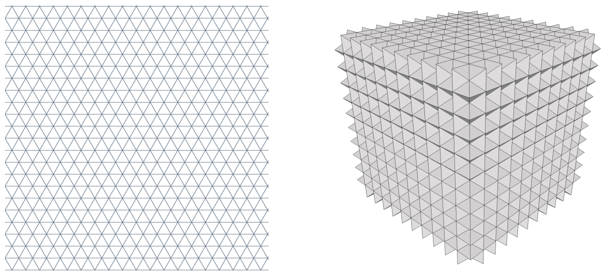


Figure 13. **Grid structure to initialize real values in 2D (left) and 3D (right).** Every face in the grid structure satisfies *Minimum-Ball* condition (Definition 3.1).

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 [37].

Multi-view Images For rendering probabilistic faces, we interpret each face's existence probability as its opacity, following [37]. We use a differentiable renderer of [37], but enhance it with anti-aliasing implemented in CUDA (Appendix 11). This renderer is built upon the efficient Gaussian Splatting renderer proposed by [14]. While it does not perform exact depth ordering between faces, it efficiently handles a large number of semi-transparent faces, making it significantly faster than renderers that perform exact depth ordering.

To compute the rendering loss, we render the mesh from multiple viewpoints and compare the rendered images to ground truth images using the L_1 loss.

8.2. Step 1: Real Value Initialization

In this step, we initialize the point-wise real values (ψ). To achieve this, we first organize the points in a regular grid, ensuring that every face in the grid satisfies the *Minimum-Ball* condition (Definition 3.1). This 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 [3]. The grids are illustrated in Fig. 13.

With these fixed points and faces, we determine which faces to include in the final mesh by minimizing the reconstruction loss L_{recon}. Specifically, for each face F in the grid, we assign an optimizable variable $\Xi(F) \in [0, 1]$, representing the face's existence probability. We then optimize $\Xi(F)$ to minimize L_{recon}. To reduce the number of redundant faces, we add a small regularization term L_{real}:

$$L_{real} = \frac{1}{|\mathbb{F}|} \sum_{F \in \mathbb{F}} \Xi(F), \qquad (15)$$

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
```

where $\mathbb{F}$ is the set of faces in the grid. The total loss optimized in this step is:

$$L = L_{recon} + \alpha_{real} \cdot L_{real}. \qquad (16)$$

We set $\alpha_{real} = 10^{-4}$.

After optimization, we collect faces with probabilities 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. We then gather the points on the remaining faces, setting their real values (ψ) to 1, while setting the real values of other points to 0.

For 2D point cloud reconstruction, this process can be accelerated by ignoring faces that do not overlap significantly with the given point cloud. However, this approach is not applicable to 3D multi-view reconstruction.

8.3. 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 nearest neighbor 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 determine 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. In this step, L consists of the reconstruction loss L_{recon} and the triangle quality loss L_{qual} (for $d = 3$). We calculate L_{qual} as described in [37]. The total loss L is defined as:

$$L = L_{recon} + \alpha_{qual} \cdot L_{qual}, \tag{17}$$

  where we set $\alpha_{qual} = 10^{-3}$.
- **Lines 19-20:** Finally, we update the point positions $\mathbb{P}$ to minimize L and iterate the process.

8.4. Step 3: Point Reduction

In this step, we remove redundant points and faces using the *Reinforce-Ball* algorithm (Sec. 4.2) for $d = 2$. For the formal definition of the algorithm, refer to Appendix 9.1.

8.5. Step 4: 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). Specifically, we optimize the face-wise probabilities using the same loss function and derive the point-wise real values from the optimized face-wise probabilities. In case of 3D multi-view reconstruction, we also conduct visibility test and remove faces that are not visible. After optimization, we optionally remove non-manifold structures, as described in Sec. 4.3, if desired by the user. If this is the last epoch, we return the post-processed mesh.

It is important to note that there may be faces that satisfy the tessellation function in Sec. 3.1 but are not included in the final mesh. This occurs because face-wise probabilities are optimized at this step, and non-manifold structures are removed based on face-wise existence.

8.6. Step 5: Subdivision

When $d = 3$, we subdivide the mesh by inserting additional points on the current faces (Sec. 4.3, Appendix 10). However, for the 2D point cloud reconstruction task, we found that starting with a very dense grid eliminates the need for additional subdivisions.

9. Details about *Reinforce-Ball* algorithm

In the *Reinforce-Ball* algorithm, we select possible face combinations within the local neighborhood and then optimize the per-point probabilities (ϕ) to minimize the number of points while preserving geometric details in 2D. Importantly, face combinations are determined **locally**, based on the *Minimum-Ball* condition, rather than **globally**. This local approach is the key to effectively removing redundant points and faces.

9.1. Algorithm

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 $B = 1024$.

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:     $\mathbb{F} \leftarrow$ Update-Query-Faces$(\mathbb{P}, \Psi)$
7:     $B_{\mathbb{F}} \leftarrow$ Compute-Minimum-Ball$(\mathbb{P}, \mathbb{F})$
8:     $j \leftarrow 0$
9:     while $j < n_1$ do
10:        ($k$ = 1, ..., $B$)
11:        $\mathbf{P}^k \leftarrow$ Sample-Points$(\mathbb{P}, \Phi)$
12:        $\mathbf{F}^k \leftarrow$ Get-Exist-Faces$(\mathbf{P}^k, \mathbb{F}, B_{\mathbb{F}})$
13:        $L_{rl}^k \leftarrow$ Compute-Loss$(\mathbb{P}, \mathbf{P}^k, \mathbf{F}^k)$
14:        $\frac{\partial \mathbb{E}[L_{rl}]}{\partial \Phi} \leftarrow$ Estimate-Gradient$(\Phi, \mathbf{P}^k, L_{rl}^k)$
15:        $\Phi \leftarrow$ Update-Gradient$(\Phi, \frac{\partial \mathbb{E}[L_{rl}]}{\partial \Phi})$
16:    end while
17:    $\mathbb{P}, \Psi \leftarrow$ Get-Remaining-Points$(\mathbb{P}, \Psi, \Phi)$
18: end while
```

- **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 function is the same as the one used in the Point Optimization step (Appendix 8.3).
- **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. (7). 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 Sec. 4.2.
- **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 [25]:

$$\nabla_\Phi \mathbb{E}_{\mathbf{P}\sim\Phi}[L_{rl}] \approx \frac{1}{B}\sum_{i=1}^{B} \nabla_\Phi \log P(\mathbf{P^i}|\Phi) \cdot L_{rl}^i. \quad (18)$$

  To reduce the variance of the gradient, we normalize L_{rl} across the batch before the computation [9].
- **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.

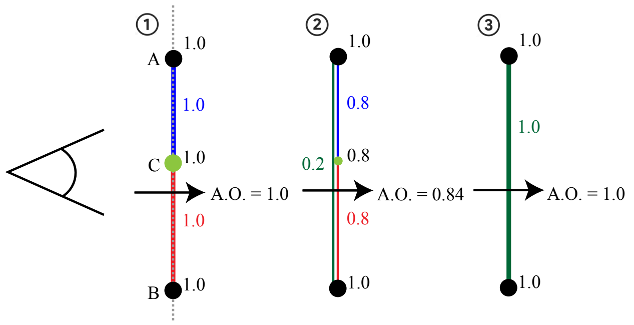


Figure 14. **Local minima of weight regularization in a rendering setting.** (1) The ground truth geometry is rendered in gray dotted line. There are 3 points (A, B, C), where only the end points (A, B) are necessary for fully representing the underlying shape. Every point has weight 1.0, which is written 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.

9.2. Local minima of Weight Regularization

As briefly discussed in Sec. 4.2, the previous approach [37], which relies on "weight regularization," cannot achieve adaptive resolution by removing redundant faces alone. Here, we explain the reasons in detail, assuming that per-point probabilities are optimized, and the *Minimum-Ball* condition is used for computing face probabilities.

In Fig. 14, 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:

$$\text{Accumulated Opacity: } 0.2 + (1.0 - 0.2) \cdot 0.8 = 0.84. \quad (19)$$

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 be:

$$0.2 + (1.0 - 0.2) \cdot 1.0 = 1.0. \quad (20)$$

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 proposed the *Reinforce-Ball* algorithm in Sec. 4.2.

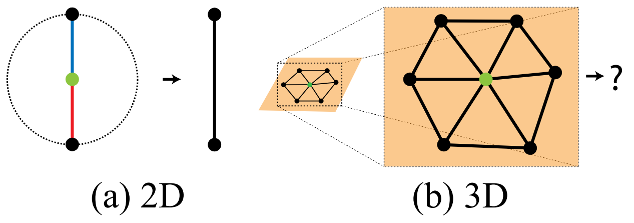


Figure 15. ***Reinforce-Ball* algorithm in (a) 2D and (b) 3D.** (a) In 2D, we can detect redundant point easily, which is rendered in green. (b) In 3D, we cannot detect such points readily, because we cannot predict how connectivity would change when we remove them.

9.3. *Reinforce-Ball* in 3D

Despite the success of the *Reinforce-Ball* algorithm in 2D, we found it difficult to extend this approach to 3D. In Fig. 15, we visualize the reasoning behind this challenge. When $d = 2$, redundant points usually lie within the inner part of a face with small curvature (i.e., a straight line segment). In this case, our algorithm can easily identify the redundant points, as removing such a point simply extends the face into a longer straight line segment.

However, when $d = 3$, it is not as straightforward to identify redundant faces. For example, in the right illustration of Fig. 15, the middle green point appears to be redundant, as it lies at the center of the points forming a planar mesh. However, we cannot predict how the connectivity will change when this point is removed. Due to this limitation, we might mistakenly predict that removing the point would create a hole. This occurs because the number of possible connections in 3D is significantly larger than in 2D. As a result, the *Reinforce-Ball* algorithm is less effective at removing redundant points and faces in 3D compared to the 2D case.

10. Details about Mesh Subdivision

In Sec. 4.3, we briefly discussed how the current mesh is subdivided by inserting additional points onto the existing faces. When additional points are added to these faces, we set the real value (ψ) of the new points to 1, ensuring that the newly created faces exist in the mesh at the start of the next epoch.

However, 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. 4, Appendix 8.5), face-wise probabilities are optimized. After optimization, we may observe faces with a probability of 0.0, while 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

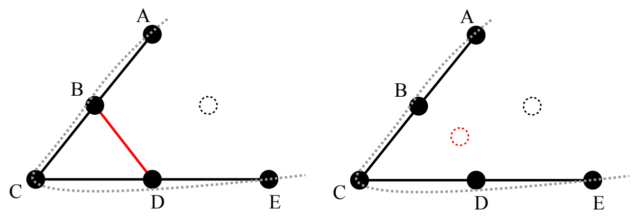


Figure 16. **Point insertion for removing undesirable face.** (Left) To reconstruct the ground truth shape, we need to set the real value (ψ) of points A-E to 1. The point rendered with dotted line has real value of 0. Then, we observe unnecessary face $\overline{BD}$ exists. (Right) To remove this face, we insert additional point that carries 0 real value near the unnecessary face.

in Fig. 16.

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. However, most of these holes are resolved during subsequent optimization steps.

11. Details about Differentiable Renderer: Anti-Aliasing

Here we provide implementation details about the (face) anti-aliasing method introduced in Sec. 4.4. 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. 17, 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:

$$\rho(F, P) = \frac{A(F, P)}{A(P)}. \tag{21}$$

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:

$$\alpha(F, P) = \alpha(F) \cdot \rho(F, P) \leq \alpha(F). \tag{22}$$

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

In Fig. 17, 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 counterclockwise 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

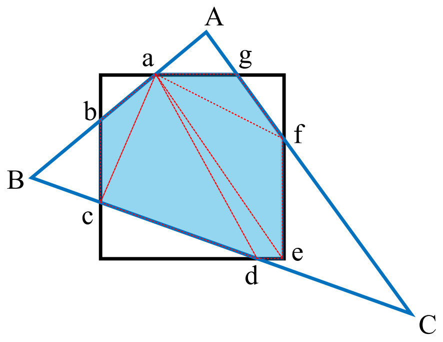


Figure 17. **Computing the overlapping area between a triangle and a pixel for anti-aliasing.**

vertices of the overlapping polygon are stored in counterclockwise 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.

To accelerate this process, we implemented the algorithm in CUDA.

12. Experimental Details

In this section, we provide details about experimental settings of the results in Sec. 5.

12.1. 2D Point Cloud Reconstruction

12.1.1. Font

Dataset For the font reconstruction experiments in Sec. 5.2, we used four publicly available font styles: Pacifico, Permanent-Marker, Playfair-Display, and Roboto. These fonts were downloaded from the Google Fonts [4] service.

Sampling Density As discussed in Sec. 5.2, we extracted 26 uppercase letters from each of these font styles and sampled dense point clouds as input (Fig. 4). To determine an appropriate point cloud density for good reconstruction quality, we considered the density of the initial triangular grid (Appendix 8.2, Fig. 13).

For a given character, we randomly sampled 1000 points per Bézier curve in the character and merged the samples. The resulting point cloud was normalized so that all points resided within a unit square $[-1, 1]^2$. The initial triangular grid covered this domain. Let the edge length of the triangular grid be x. Next, we divided the domain into a *rectangular* grid with an edge length approximately $x/2$, giving the rectangular grid a slightly higher resolution than the triangular grid.

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

For each cell in the rectangular grid, if multiple sample points were present within the cell, we randomly selected one point to retain. This rectangular grid acted as a filter to scale down the point cloud while preserving sufficient density for shape reconstruction. Since the rectangular grid had a higher resolution than the triangular grid, this approach provided a point cloud dense enough to reconstruct the shapes effectively.

To summarize, the density of the sample point cloud depends on the edge length of the initial triangular grid. For font reconstruction, we set the edge length to 0.005.

However, sparse regions lacking sample points occasionally resulted in holes in the reconstructed meshes. To address this issue, we can either sample more points (e.g., more than 1000 points per Bézier curve) or use a finer rectangular grid. Nevertheless, we found that our current configuration generally performed well.

DMesh Settings For comparison, we used DMesh [37]. The following experimental settings were used for font reconstruction:

- **Step 1: Real Value Initialization**
  - Number of optimization steps: 100
  - Learning rate: 0.3
- **Step 2: Point Position Optimization**
  - Number of optimization steps: 3000
  - Warm-up steps: 500 [5]
  - Learning rate: 0.001

DMesh++ Settings In general, the experimental settings of DMesh++ are the same as those of DMesh. The only difference is that DMesh++ optimizes point positions for 500 steps in Step 2, the same as the number of warm-up steps in DMesh. After this, DMesh++ proceeds to Step 3, described in Appendix 9.1. For this step, the learning rate for point probabilities was set to 0.01.

12.1.2. Complex Drawings

Dataset We used six vector graphic images representing complex drawings (Figs. 1, 11 and 20 to 23). These images were downloaded from Adobe Stock [6].

Sampling Density For complex drawings, we required more sample points compared to fonts due to their increased complexity. Therefore, we set the edge length of the initial triangular grid to 0.001, which is significantly smaller than the 0.005 used in the font experiments.

For these inputs, we could not use DMesh for reconstruction due to memory limitations. For DMesh++, we did not apply the *Reinforce-Ball* algorithm for adaptive resolution due to its excessive computational cost. All other settings remained the same as those used in the font experiments.

[5]Number of optimization steps before starting weight regularization (Sec. 5.2). If weight regularization is not used, optimization ends after these warm-up steps.

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

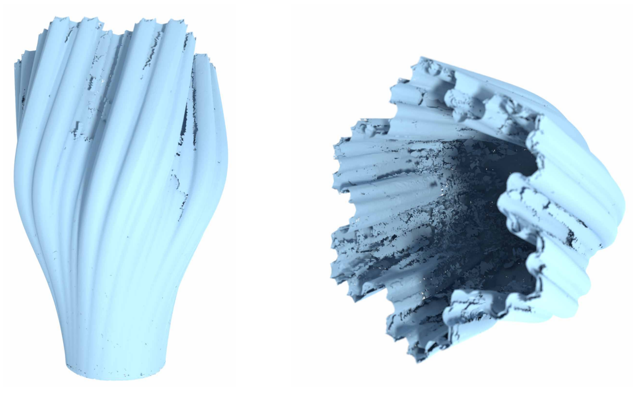


Figure 18. **False floaters in 3D reconstruction result for a complex vase.** Image captured from front (left), and top (right) of the reconstructed mesh. When there are many self-occlusions that hinder dense observation, false floaters are created near the surface.

12.2. 3D Multi-View Reconstruction

12.2.1. Thingi10K

Dataset As mentioned in Sec. 5.3, we selected 10 closed surfaces and 10 open surfaces from the Thingi10K dataset [46] to evaluate the 3D multi-view reconstruction quality. 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.

We selected these models because they exhibit minimal self-occlusions, allowing for dense observations from multi-view images. In contrast, models with significant self-occlusions, such as the vase shown in Fig. 18, do not meet this criterion. The vase's rugged surface structure leads to numerous self-occlusions, and its inner surface is not well-observed from most input images. Consequently, our algorithm struggles to effectively remove false inner parts in such cases.

It is worth noting that the primary focus of our paper is to demonstrate the effectiveness of DMesh++ as an efficient shape representation, rather than to propose a perfect reconstruction algorithm. Consequently, we assume perfect observations for reconstructing the given shapes and exclude models with significant self-occlusions from our dataset. We believe this limitation could be addressed by introducing certain topological assumptions, as discussed in Sec. 6.

For all selected models, we normalized their size to fit within a unit cube $[-1, 1]^3$ and captured diffuse and depth images with a resolution of 512×512 pixels from 64 viewpoints (Fig. 4).

Next, we describe the experimental settings for each method used for comparison in Sec. 5.3.

Remeshing Settings For Remeshing [29], we used the following experimental settings:

- 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 Settings For DMTet [33], we used the following experimental settings:

- 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 Settings For FlexiCubes [34], we used the following experimental settings:

- 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 Settings For GShell [19], we used the following experimental settings:

- Image Batch Size: 8
- Number of Optimization Steps: 5000
- Number of Warm-up Steps: 4500
- Learning Rate: 0.01

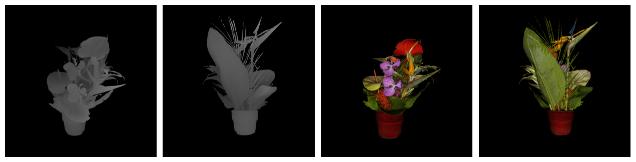


Figure 19. **Input depth maps (left) and color maps (right) for 3D multi-view reconstruction.** These images were used to reconstruct the *flowers* model in Fig. 1.

- 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 For DMesh [37], we used the experimental settings from the original implementation. The only modification was in the resolution of the input images.

DMesh++ Settings For DMesh++, we used the following experimental 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.

12.2.2. Objaverse

Dataset We used textured meshes downloaded from Objaverse [8] to test our reconstruction algorithm. We chose them based on the same conditions as Thingi10K experiments. As these meshes have textures, we replaced the input diffuse maps with color maps, as shown in Fig. 19. We provide the results in Appendix 13.3.

DMesh++ Settings To reconstruct the meshes more faithfully, we used higher resolution images for these experiments. Also, we optimized the mesh for larger number of epochs, as shown below.

- Initial Grid Edge Length: 0.05
- Learning rate (real value, ψ): 0.01
- Learning rate (position): 0.001
- Number of epochs: 4
  - Image Res. / Batch Size at Epoch 1: (256, 256), 1

	Geometric Accuacy					Mesh Quality				Statistics		
Method	CD($\times 10^{-3}$)↓	F1↑	NC↑	ECD↓	EF1↑	AR↓	SI↓	NME↓	NMV↓	# Verts.	# Faces.	Time (sec)
Remeshing [29]	0.126	0.330	0.950	0.114	0.123	1.429	0.001	0	0	13527	27049	25
DMTet [33]	0.172	0.242	0.935	0.309	0.019	6.015	0	0	0	19410	38816	201
FlexiCubes [34]	0.385	0.348	0.938	0.227	0.026	2.049	0.001	0	0.003	13637	26874	89
GShell [19]	0.145	0.339	0.942	0.246	0.031	2.662	0.013	0	0.002	14502	28624	209
DMesh [37]	0.241	0.292	0.942	0.141	0.059	1.803	0	0.001	0	2675	5348	765
DMesh++(Ours)	0.141	0.345	0.965	0.170	0.073	1.604	0	0	0	13170	25856	214

Table 3. **Quantitative comparison of 3D multi-view reconstruction results for *closed* surfaces.** We follow the notation of Tab. 2.

	Geometric Accuacy					Mesh Quality				Statistics		
Method	CD($\times 10^{-3}$)↓	F1↑	NC↑	ECD↓	EF1↑	AR↓	SI↓	NME↓	NMV↓	# Verts.	# Faces.	Time (sec)
Remeshing [29]	2.290	0.524	0.933	0.149	0.028	1.568	0.527	0	0	10263	20521	25
DMTet [33]	1.887	0.262	0.915	0.266	0.003	7.045	0.001	0	0	16517	33098	200
FlexiCubes [34]	1.752	0.362	0.911	0.291	0.013	2.083	0.030	0	0.004	11384	22378	87
GShell [19]	0.940	0.467	0.936	0.089	0.028	2.906	0.112	0	0.002	9338	17651	207
DMesh [37]	0.775	0.505	0.931	0.089	0.124	1.794	0	0.066	0.001	2086	4324	736
DMesh++(Ours)	0.105	0.575	0.964	0.026	0.169	1.603	0	0	0	6873	12744	194

Table 4. **Quantitative comparison of 3D multi-view reconstruction results for *open* surfaces.** We follow the notation of Tab. 2.

  - Image Res. / Batch Size at Epoch 2: (512, 512), 1
  - Image Res. / Batch Size at Epoch 3: (1024, 1024), 1
  - Image Res. / Batch Size at Epoch 4: (1024, 1024), 1
- Number of optimization steps
  - Step 1 (Real value initialization): 1000
  - Step 2 (Point Position optimization): 2000
  - Step 3 (Real value optimization): 1000

13. Additional Experimental Results

13.1. 2D Complex Drawing Reconstruction

In Figs. 20 to 23, we present additional reconstruction results for complex drawings in 2D. For each result, the final mesh is rendered at the top. At the bottom, we render the "real" part in blue and the "imaginary" part in gray.

The number of edges and computational time for each reconstruction are provided in the figure captions. It is important to note that the computational cost increases approximately linearly with the number of edges.

We found that the majority of the computational cost originates from computing the expected Chamfer Distance (CD) loss for reconstruction, rather than from evaluating face probabilities. Thus, we believe that optimizing the algorithm for computing the expected CD loss could significantly accelerate this process.

13.2. 3D Multi-View Reconstruction for Thingi10k

In Tabs. 3 and 4, we present quantitative comparisons of multi-view reconstruction results for closed and open surfaces, respectively. Notably, DMesh++ outperforms other methods by a large margin for open surfaces.

For closed surfaces, while the Remeshing method [29] achieved the overall best results, the performance of DMesh++ was comparable and did not lag significantly.

In summary, as shown in Tab. 2, DMesh++ achieved the best overall results across the entire dataset.

13.3. 3D Multi-View Reconstruction for Objaverse

In Figs. 1, 11 and 24 to 27, we present 3D multi-view reconstruction results for models from the Objaverse [8] dataset. These results demonstrate that our reconstruction algorithm effectively recovers a variety of shapes without significant difficulty.

As with the 2D complex drawings (Sec. 13.1), we report the number of faces and the computation time for each reconstruction in Figs. 24 to 27. It can be observed that computational cost generally increases with the number of faces and is typically higher for scenes compared to individual objects.

Finally, we note that the DNA example in Fig. 1 is a commercial model from TurboSquid [7]. Unlike the other models from Objaverse, we initialized our mesh using 10K sample points from the ground truth mesh, following [37]. This was necessary because recovering the thin structures of the DNA model proved challenging with the current coarse-to-fine approach. We believe that developing alternative optimization strategies to address such challenges would be an interesting direction for future research.

[7]https://www.turbosquid.com/

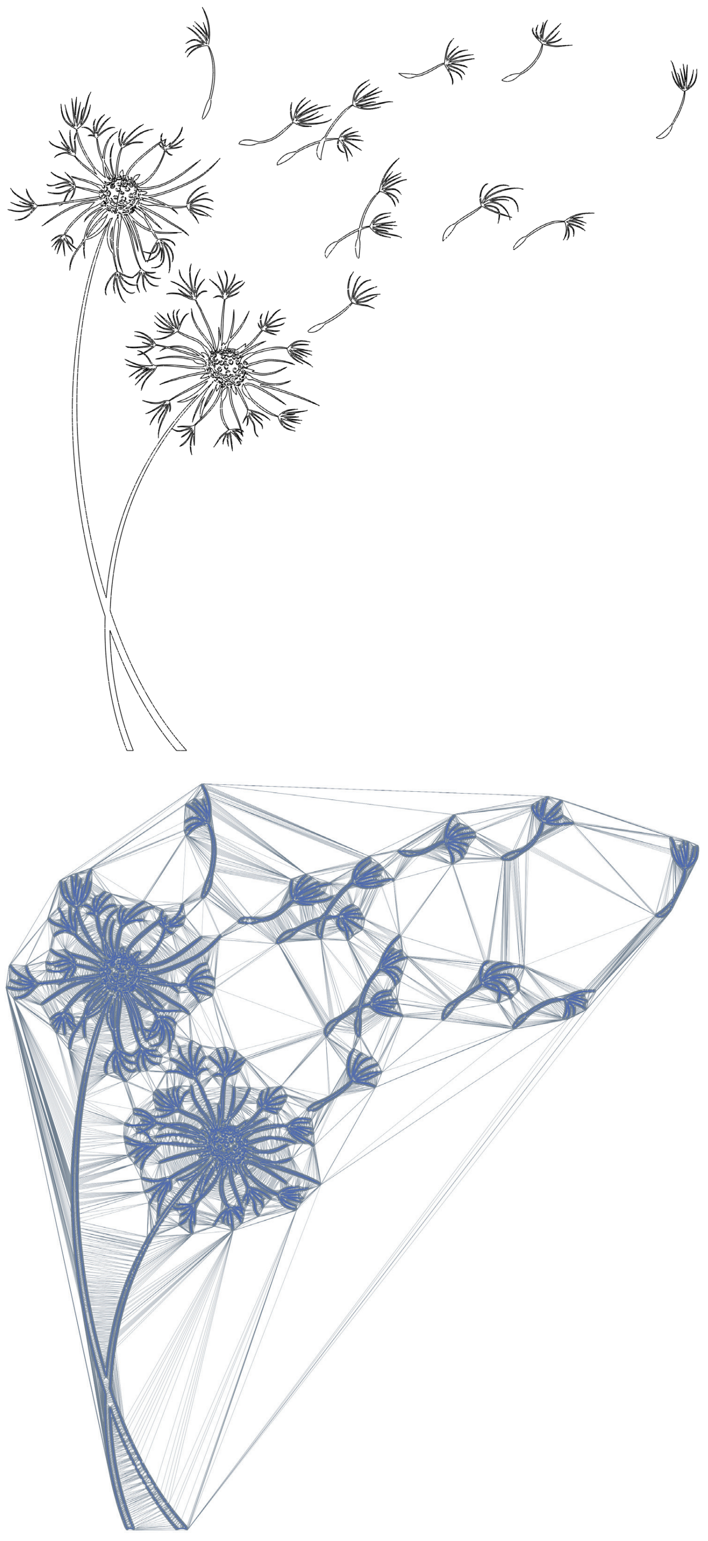


Figure 20. **2D Point Cloud Reconstruction result: Flower.** # Edge: 99K, Time: 6 min.

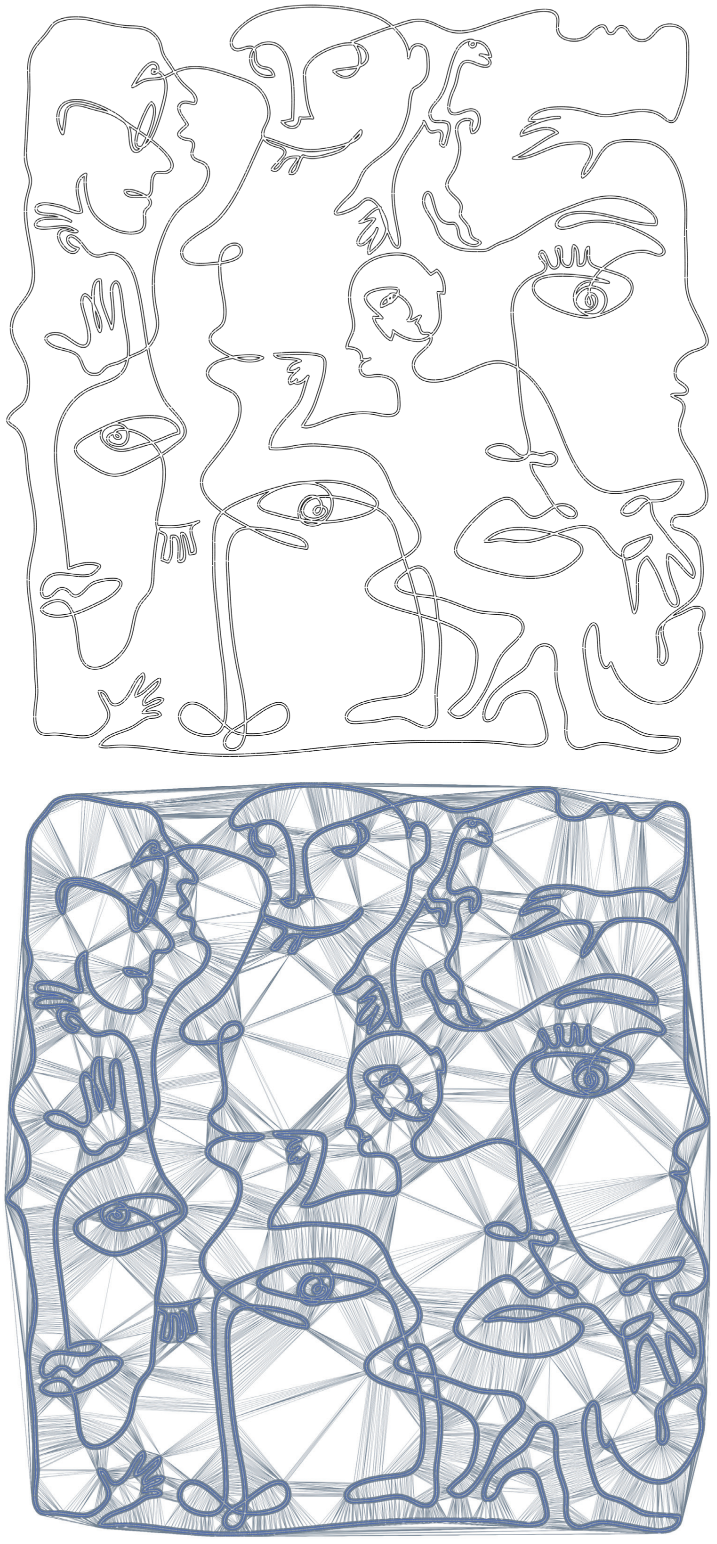
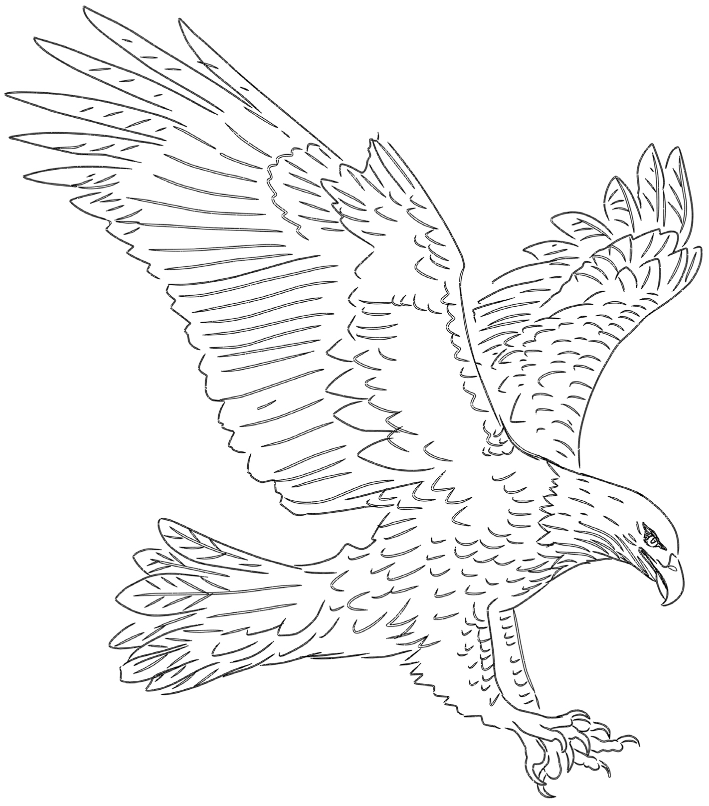


Figure 21. **2D Point Cloud Reconstruction result: Picasso Painting.** # Edge: 159K, Time: 8 min.



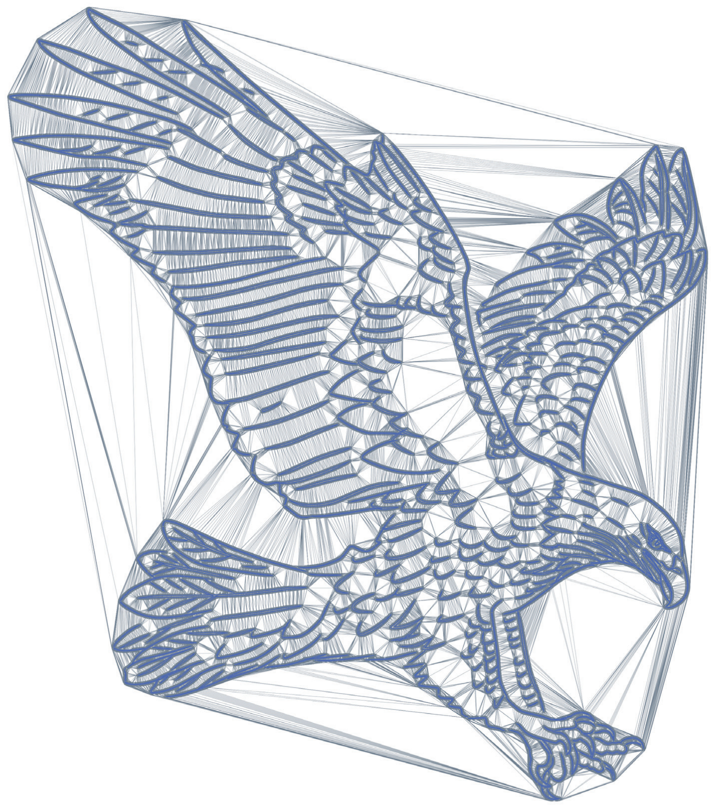


Figure 22. **2D Point Cloud Reconstruction result: Eagle.** # Edge: 179K, Time: 11 min.

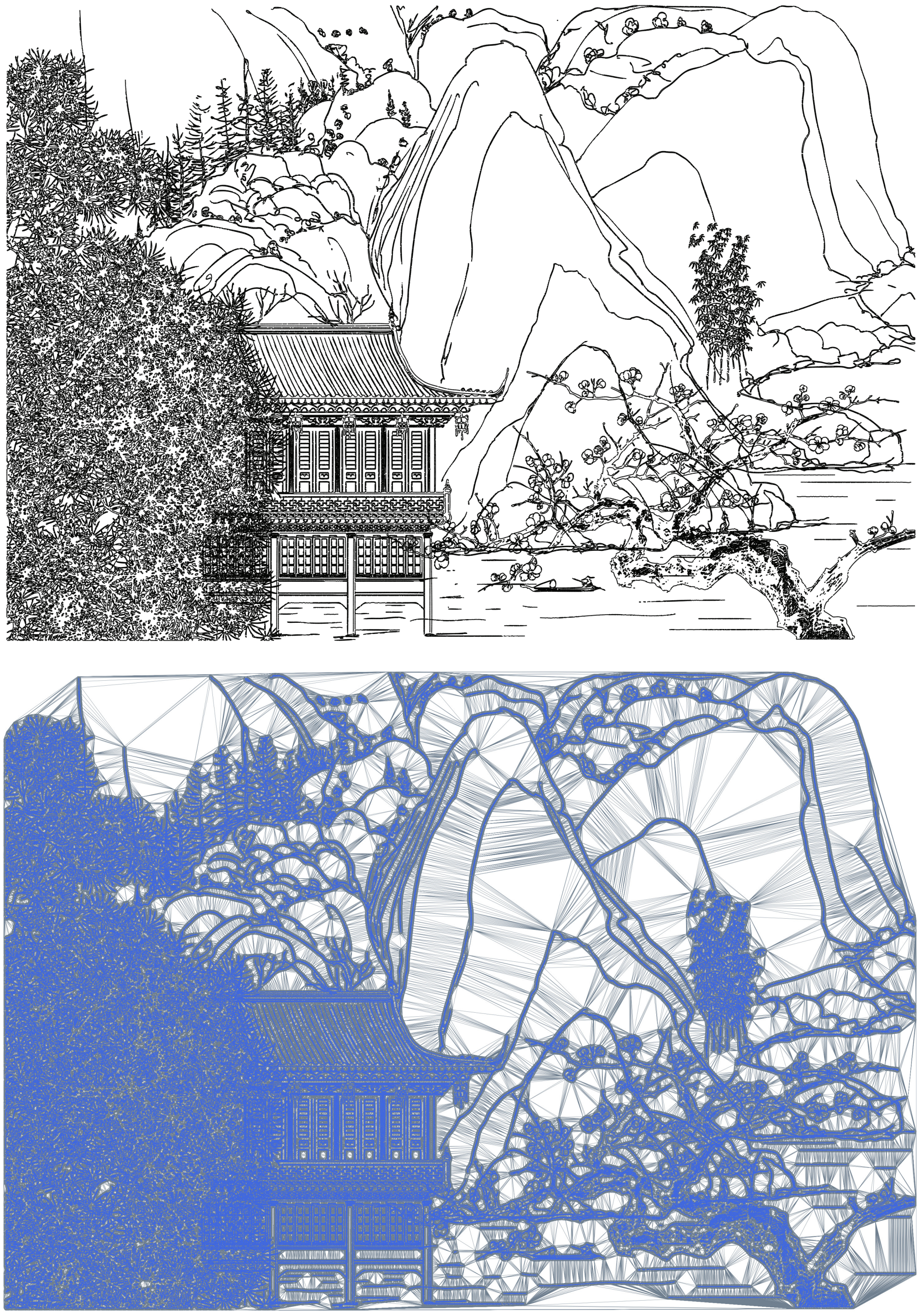


Figure 23. **2D Point Cloud Reconstruction result: Chinese Drawing.** # Edge: 987K, Time: 86 min.

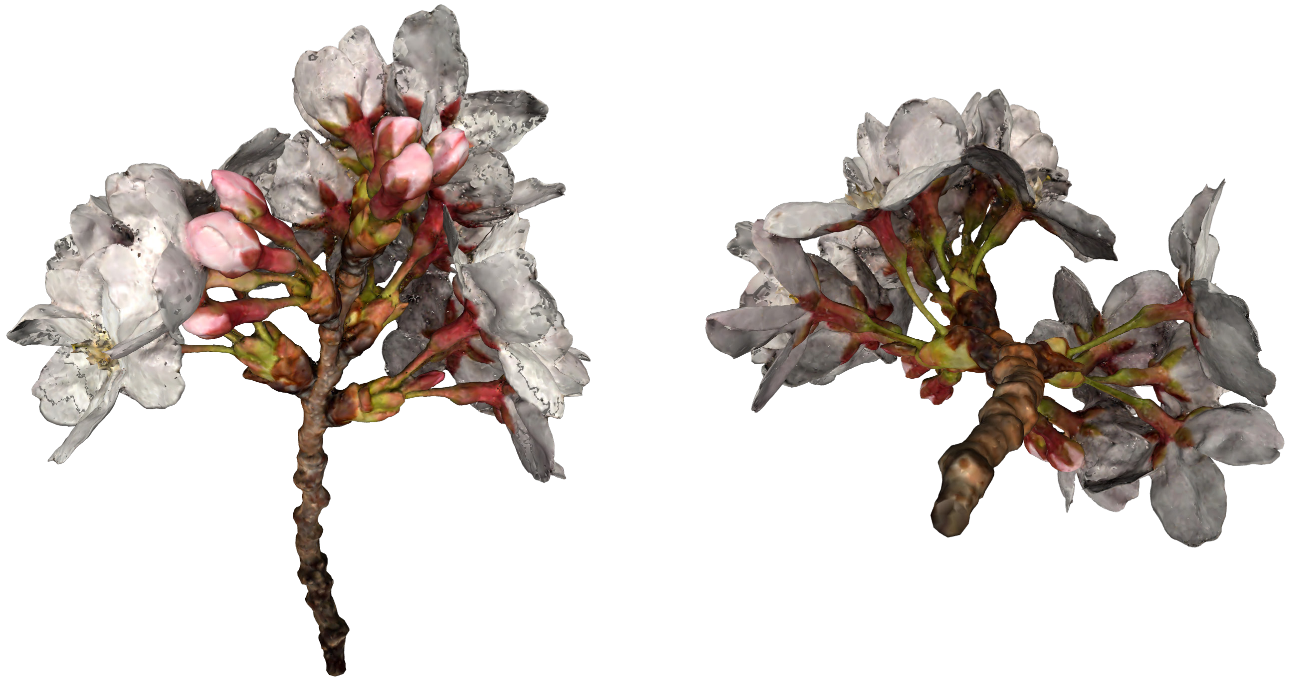


Figure 24. **3D Multi-View Reconstruction result: Sakura flower.** # Face: 296K, Time: 14 min.

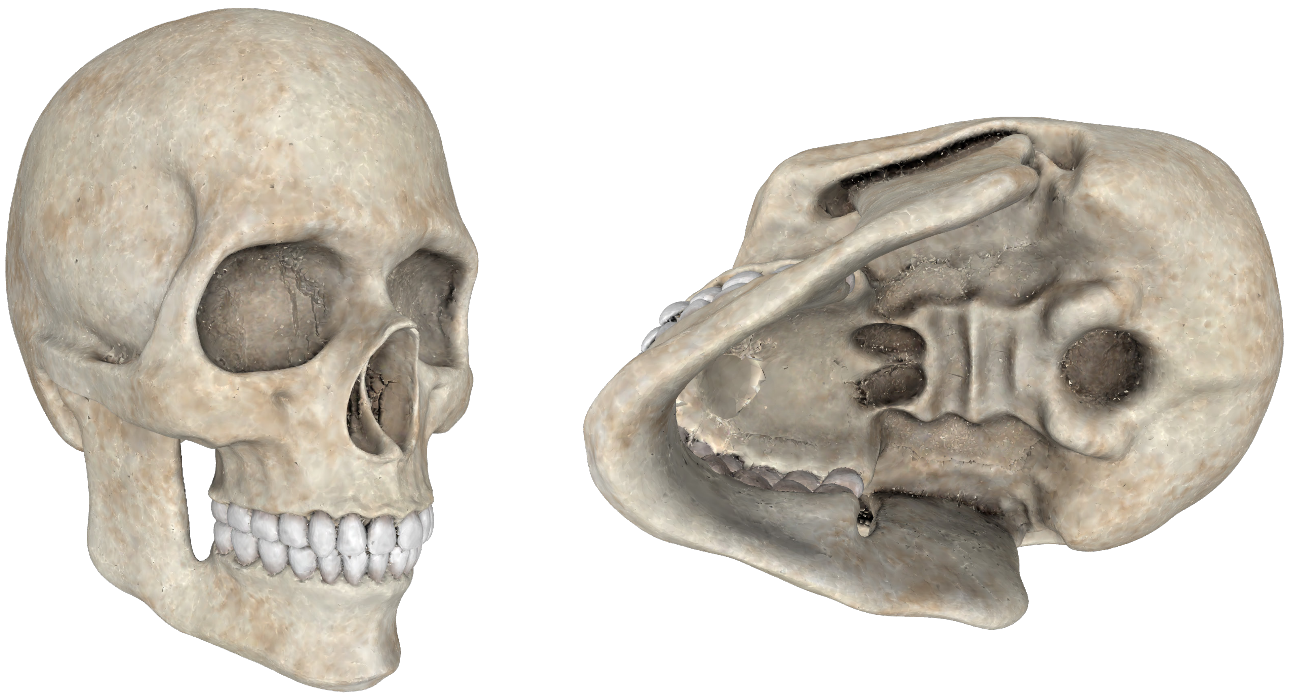


Figure 25. **3D Multi-View Reconstruction result: Skull.** # Face: 416K, Time: 16 min.

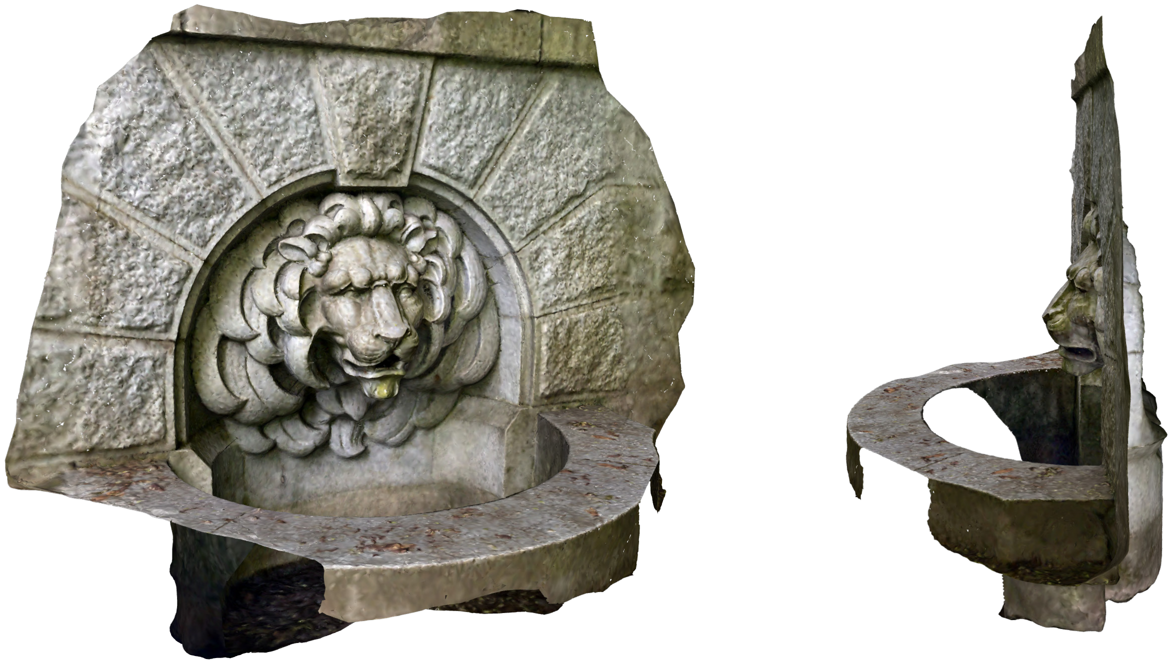


Figure 26. **3D Multi-View Reconstruction result: A fountain.** # Face: 521K, Time: 36 min.

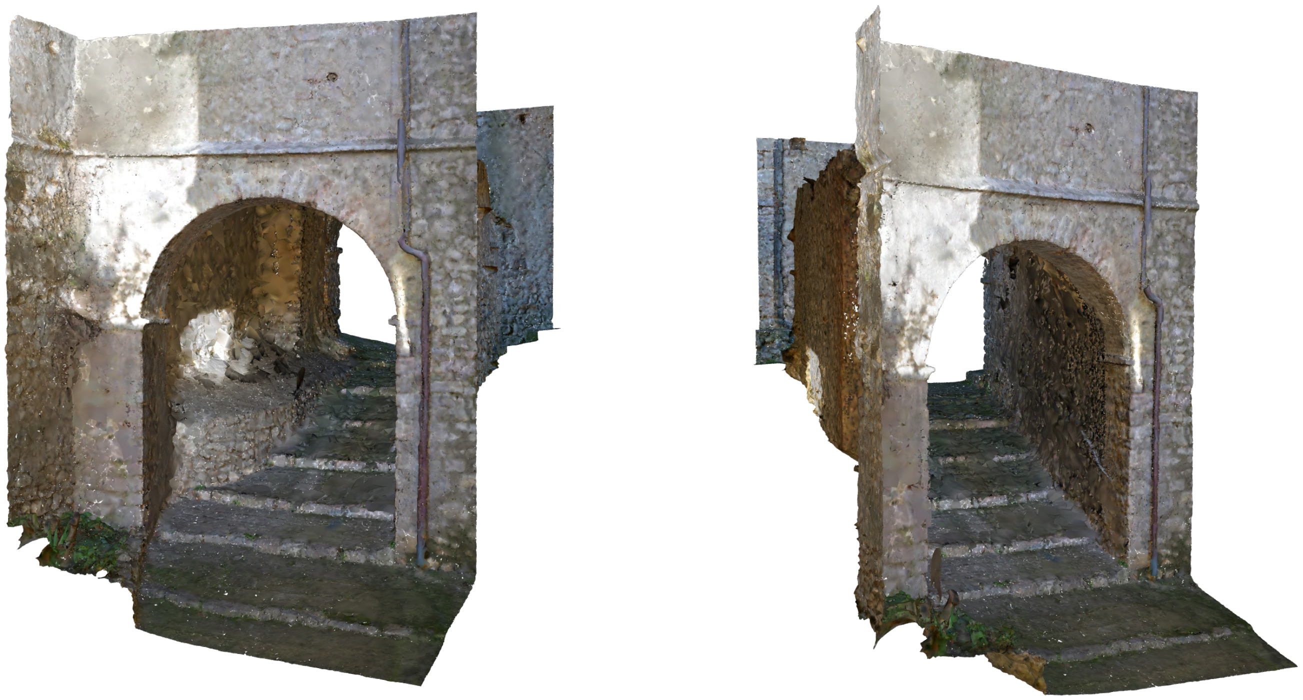


Figure 27. **3D Multi-View Reconstruction result: A scene with a stone staircase.** # Face: 353K, Time: 36 min.