DMesh++: An Efficient Differentiable Mesh for Complex Shapes

Supplementary Material

7. Details about *Minimum-Ball* algorithm

7.1. Algorithm

Algorithm 1 `Minimum-Ball`

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

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

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

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

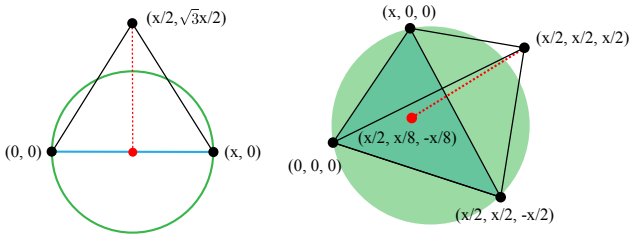


Figure 13. **Common signed distance for a 2D (left) and 3D (right) face in (initial) regular grid.** We compute the signed distance by subtracting the radius of the minimum bounding ball from the length of the red line. The red dot represents the center of the minimum bounding ball.

7.2. Minimum-Ball computation

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

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

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

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

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

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

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

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

7.3. Sigmoid coefficient α_{min}

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

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

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

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

For $d = 2$:

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

For $d = 3$:

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

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

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

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

7.4. Nearest neighbor caching

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

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

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

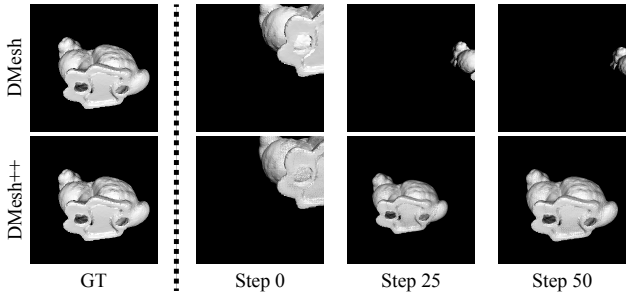


Figure 14. **Role of visibility gradient in geometric optimization.** In this experiment, we optimize the translation vector of the object by comparing its rendered image and the ground truth image on the left. Since the differentiable renderer of DMesh [42] does not implement visibility gradient, while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

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

8. Details about Reconstruction Process

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

8.1. Loss Formulation

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

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

We explain each of these terms below.

8.1.1. Reconstruction Loss (L_{recon})

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

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

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

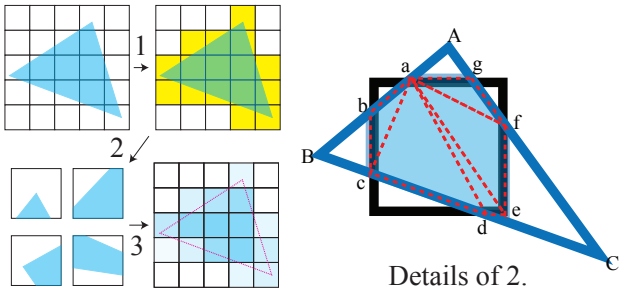


Figure 15. **Implementation of anti-aliasing in our differentiable renderer.** On the left, we show the process of anti-aliasing: 1) Find pixels that overlap with the given triangle, 2) Find the area that each pixel overlaps with the given triangle, and 3) Determine the color of each pixel based on the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

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

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

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

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

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

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

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

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

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

8.1.2. Triangle Quality Loss (L_{qual})

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

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

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

8.1.3. Real Loss (L_{real})

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

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

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

8.2. Reconstruction Steps

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

8.2.1. Step 1: Initialization

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

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

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

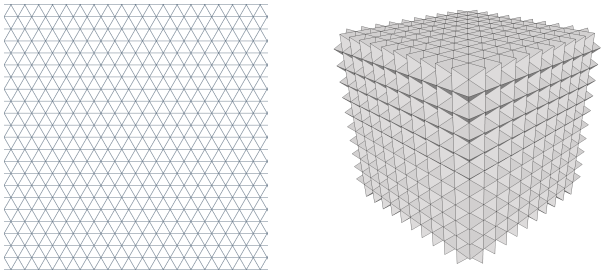


Figure 16. **Grid structure to initialize real values in 2D (left) and 3D (right).** Every face in the grid structure satisfies *Minimum-Ball* condition (Definition 3.1).

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

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

8.2.2. Step 2: Position Optimization

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

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

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

Algorithm 2 `Position Optimization`

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

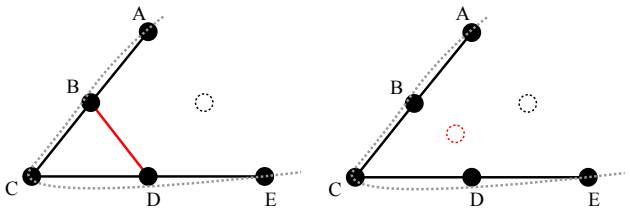


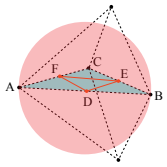
Figure 17. **Point insertion for removing undesirable face.** (Left) To reconstruct the ground truth shape, we need to set the real value (ψ) of points A-E to 1. The point rendered with dotted line has real value of 0. Then, we observe unnecessary face $\overline{BD}$ exists. (Right) To remove this face,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

8.2.3. Step 3: Real Value Optimization

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

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

8.2.4. Step 4: Subdivision



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

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

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

9. Experimental Details and Additional Results

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

9.1. Dataset

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

9.1.1. Font

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

9.1.2. Thingi10K

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

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

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

9.1.3. Objaverse

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

9.2. 2D Point Cloud Reconstruction

9.2.1. Hyperparameters

DMesh++

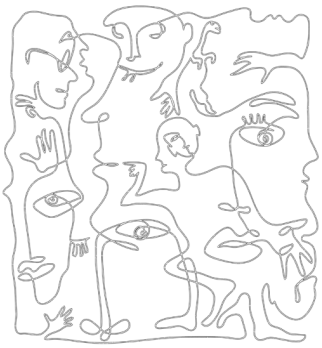
- Learning rate (real value, ψ): 0.3
- Learning rate (position): 0.001
- Number of epochs: 1
- Number of optimization steps
  - Step 1 (Real value initialization): 100
  - Step 2 (Point position optimization): 500

9.2.2. Reconstruction of Complex Drawings

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

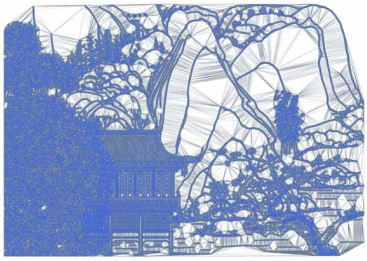
(a) Flower, # Edge = 99K, 6 min.

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



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

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



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

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

9.3. 3D Point Cloud Reconstruction

9.3.1. Hyperparameters

DMesh++

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

9.4. 3D Multi-View Reconstruction

9.4.1. Hyperparameters

Remeshing [33]

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

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

DMTet [38]

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

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

FlexiCubes [39]

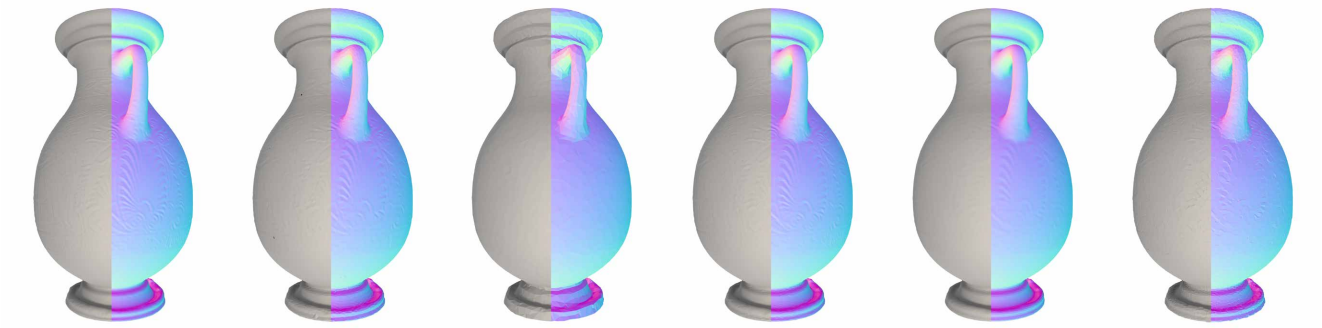
- Image Batch Size: 8
- Number of Optimization Steps: 2000
- Number of Warm-up Steps: 1500
- Learning Rate: 0.01
- Grid Resolution: 80
- Triangle Aspect Ratio Loss Weight: 0.01

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

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

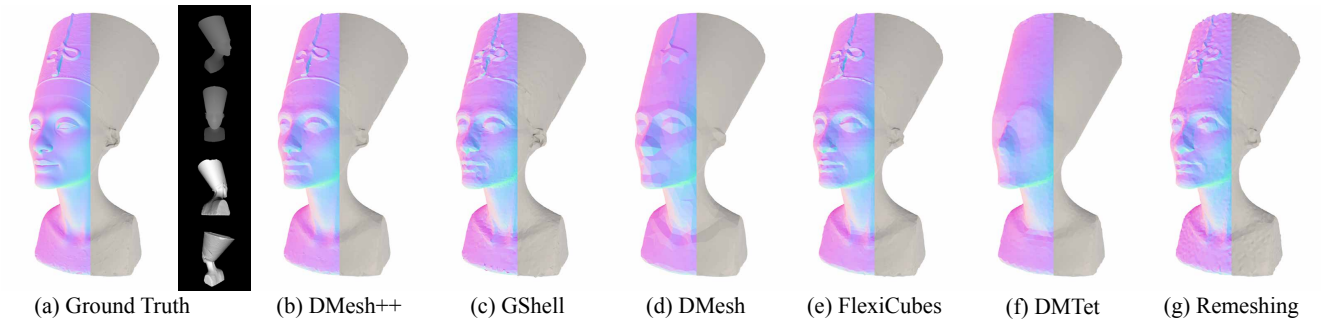
GShell [23]

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



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

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



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

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

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

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

DMesh++ Settings

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

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

9.4.2. Limitations

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

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

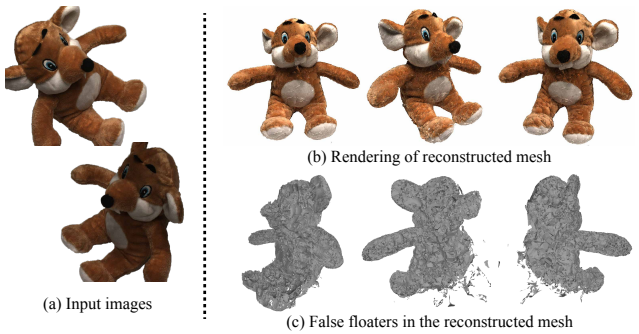


Figure 21. **3D reconstruction from real-world images in DTU dataset [15].** The input images are shown on left, and the reconstructed mesh is shown on right.

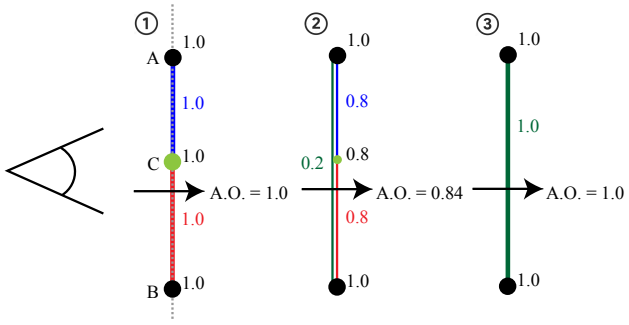


Figure 22. **Local minima of weight regularization in a rendering setting.** (1) The ground truth geometry is rendered in gray dotted line. There are 3 points (A, B, C), where only the end points (A, B) are necessary for fully representing the underlying shape. Every point has weight 1.0, which is written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

10. *Reinforce-Ball* algorithm

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

10.1. Local Minima of Weight Regularization

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

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

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

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

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

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

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

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

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

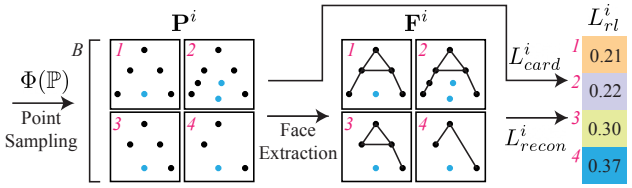


Figure 23. **Overview of *Reinforce-Ball* Algorithm.** Based on per-point existence probability ($\Phi(\mathbb{P})$), we sample points for B number of batches ($\mathbf{P}^i$). Here we use $B = 4$, and assume we are reconstructing shape "A". The points with $\psi = 1$ are rendered in black, while those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

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

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

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

10.2. Algorithm Overview

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

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

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

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

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

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

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

Algorithm 3 `Reinforce-Ball`

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

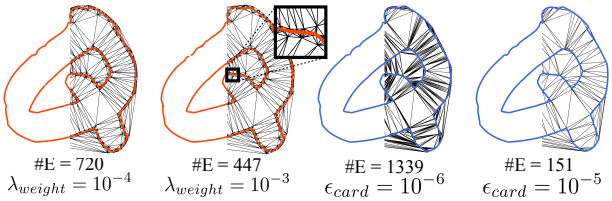
10.3. Formal Definition

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

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

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

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



(a) DMesh (b) DMesh++

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

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

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

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

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

10.4. Experimental Results

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

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

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