

# MeshRipple: Structured Autoregressive Generation of Artist-Meshes

## Supplementary Material

### A. Ripple Tokenization Algorithm

---

**Algorithm 1:** Ripple Tokenization

---

**Input** : Mesh  $\mathcal{M}$   
**Preprocess**:  $\mathcal{M} \rightarrow$  Half Edges  $\mathbf{H} = \{\mathbf{h}_i\}_N$   
**Output** : Sequence  $\mathcal{S}$ , Root Index  $\mathbf{R}$   
**Data**: FIFO Frontier Queue  $\mathcal{B}$

```

 $\mathcal{S}.\text{append}(\text{BOS});$ 
for HalfEdge  $\mathbf{h}$  in  $\mathbf{H}$ :
    if  $\mathbf{h}.\text{face}.\text{vis}$ :
        continue;
     $\mathbf{h}.\text{face}.\text{vis} = \text{true};$ 
     $\mathcal{B}.\text{enqueue}(\mathbf{h});$ 
     $\mathcal{S}.\text{append}(\mathbf{N});$ 
    AddFace ( $\mathcal{S}$ ,  $\mathbf{h}$ );
    while  $\mathcal{B}$  is not empty:
         $\mathbf{h} = \mathcal{B}.\text{dequeue}();$ 
        for HalfEdge  $\mathbf{h}_o$  in [ $\mathbf{h}.\text{p.o}$ ;  $\mathbf{h}.\text{n.o}$ ;  $\mathbf{h}.\text{o}$ ]:
            if not  $\mathbf{h}_o.\text{face}.\text{vis}$ :
                 $\mathbf{h}_o.\text{face}.\text{vis} = \text{true};$ 
                AddFace ( $\mathcal{S}$ ,  $\mathbf{h}_o$ );
                RecordRoot ( $\mathbf{R}$ ,  $\mathbf{h}$ ,  $\mathbf{h}_o$ );
                 $\mathcal{B}.\text{enqueue}(\mathbf{h}_o);$ 
    Project  $\mathcal{S}$  to vertex coordinates
Output  $\mathcal{S}$ ,  $\mathbf{R}$ 

```

---

In this section, we elaborate on the proposed Ripple Tokenization. We begin by preprocessing the input mesh: vertices are quantized into discrete bins (set to 256 in our experiments), duplicated vertices are merged, and degenerate faces are removed. This yields a sanitized mesh  $\mathcal{M} = \{\mathcal{F}, \mathcal{V}\}$ , where each face  $\mathbf{F}_i \in \mathcal{F}$  is defined as  $\mathbf{F}_i = [\mathbf{a}_0^i, \mathbf{a}_1^i, \mathbf{a}_2^i]$ , containing the indices of the vertices forming the face. To establish a deterministic traversal order, we sort all faces lexicographically based on their vertex coordinates (specifically using  $z$ - $y$ - $x$  ordering). Subsequently, we construct a half-edge data structure for the mesh, respecting the orientation defined by the face normals. The tokenization process employs a breadth-first traversal initialized at the first half-edge of the lexicographically first face,  $\mathbf{F}_0$ . During each iteration, a half-edge  $\mathbf{h}$  is dequeued, and we sequentially access the twin half-edges of its predecessor ( $\mathbf{h}.\text{p}$ ), its successor ( $\mathbf{h}.\text{n}$ ), and the half-edge itself ( $\mathbf{h}$ ). This sequence strictly defines the traversal priority for the neighbors of any given face  $\mathbf{F}_i$ . To prevent cy-

cles and redundancy, we maintain a binary visitation state,  $\text{vis}$ , for each face. When a neighboring face is accessed via a twin half-edge, we check its state; if it has not yet been visited, we mark  $\text{vis}$  as true and add the corresponding half-edge to the queue. This propagation continues until all reachable half-edges in the connected component have been processed.

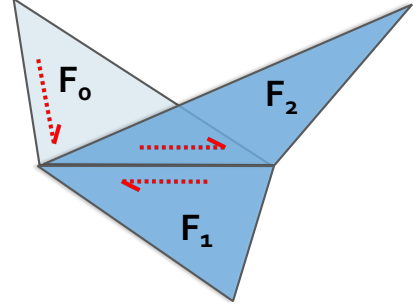


Figure 1. **Illustration of Non-Manifold Traversal Strategy.** Direct traversal between  $F_0$  and  $F_2$  is inhibited because their incident half-edges are co-directional. The algorithm resolves this by extending traversal through the intermediate face  $F_1$ , maintaining normal vector consistency.

**Non-manifoldness.** To ensure consistent surface orientation, our algorithm restricts traversal across non-manifold edges to incident half-edges with opposite orientations. Consequently, traversal is explicitly inhibited when an adjacent half-edge is co-directional with the current half-edge. As illustrated in Figure 1, although faces  $\mathbf{F}_0$  and  $\mathbf{F}_2$  share identical normal vectors, the co-directionality of their shared boundary prevents direct propagation between them. However, structural connectivity allows for indirect traversal. In the example shown, the algorithm successfully propagates from  $\mathbf{F}_0$  to  $\mathbf{F}_1$ , provided their interface satisfies the orientation constraint (opposite half-edges). From  $\mathbf{F}_1$ , the traversal can subsequently extend to  $\mathbf{F}_2$ . Crucially, in scenarios where a face is effectively isolated by co-directional half-edges on all sides—indicating an irresolvable conflict in face normals—we interpret this as a topological discontinuity and segregate the regions into separate connected components.

**Disconnected Components.** For each independent connected component, the starting face lacks a preceding frontier face. Consequently, we assign a special token **N** to explicitly mark the initiation of a new component, as illustrated in Figure 2.

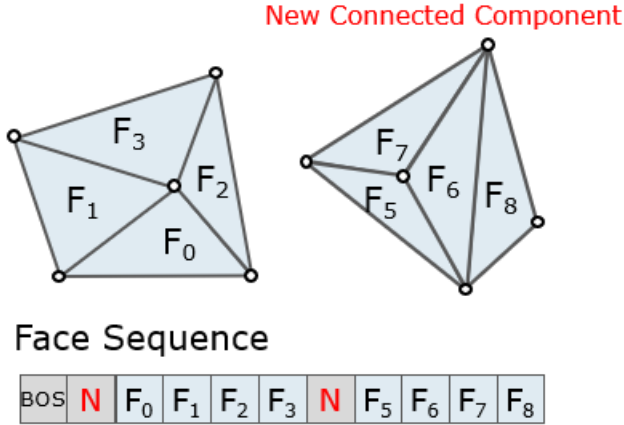


Figure 2. **Initialization of Disconnected Components.** When initiating traversal for a topologically disjoint component, the starting face lacks a preceding frontier face. A special identifier  $N$  is used to explicitly mark this boundary.

## B. More Implementation Details

### B.1. Filtering Training Data

Table 1. **Model Architecture Hyperparameters.**

	Small scale	Large scale
Parameter count	500 M	800M
Batch Size	8	4
Hourglass	4,4,9	4,4,9
Heads	10	16
$d_{model}$	768	1024
$d_{FFN}$	3072	4096
Learning rate	$1e-4$	$1e-4$
LR scheduler	Cosine	Cosine
Weight decay	0.1	0.1
Gradient Clip	1.0	1.0

### B.2. More Training Details

The significant variance in data quality across collected datasets poses a challenge for robust model training. To address this, we implement a strict filtering pipeline based on geometric topology and quantization artifacts.

**Geometric and Topological Filtering.** We discard meshes with face counts outside the range  $[500, 20k]$ , vertex-to-face ratios  $> 0.8$ , or where narrow faces (minimum angle  $< 5^\circ$ ) exceed 20% of the total. To ensure topological simplicity, we exclude meshes exceeding a maximum BFS displacement of 100, a boundary length of 500, or 20 connected components. The component count is determined after pruning small clusters ( $< 10$  faces) located within a normalized distance of 0.05 from valid components.

**Vertex Merge Filtering.** To mitigate artifacts from discrete vertex merging, we filter meshes where the vertex count drops by  $> 50\%$ , or where newly introduced self-intersecting or overlapping faces exceed 10%. Self-intersections are detected via `trimesh`, while overlapping faces are identified by grouping coplanar faces and verifying candidates via the Separating Axis Theorem (SAT).

We implemented two variants of MeshRipple: a small-scale model and a large-scale model. The detailed architectural specifications are presented in Table 1. It is worth noting that the reported parameter counts include the parameters of Michelangelo encoder. Furthermore, to optimize GPU memory usage, we utilized bfloat16 (bf16) precision and FlashAttention.

Furthermore, we employ point cloud noise injection, rotation, and scaling for data augmentation. Specifically, during point cloud sampling, Gaussian noise is added to the vertex coordinates with a probability of 0.5. For rotation, the mesh is set to perform rotational augmentation around the  $z$ -axis, with each unit being  $30^\circ$ , covering a total of  $360^\circ$ . Finally, for scaling, we uniformly sample a scaling factor from the range  $[0.75, 1.25]$  for each axis.

## C. More Ablation Study

Table 2. **Ablation study on the truncation window size.** Metrics are evaluated on 160 dense meshes.

Window Size	CD ( $\downarrow$ )	HD ( $\downarrow$ )	NC ( $\uparrow$ )
1k	0.051948	0.109130	0.796585
2k	<b>0.050651</b>	<b>0.104539</b>	<b>0.798088</b>

**Truncation Window Size.** We conduct an ablation study on the truncation window size, comparing 1k and 2k, on a dataset of 100k samples. The evaluation is performed on 160 dense meshes, with results presented in Table 2.

**Resources w/o NSCA.** Table 3 compares the computational resource consumption with and without NSCA across various configurations. We have noticed the advantages of NSCA in terms of time and GPU memory resource consumption in long sequences, which can help us extend MeshRipple to longer window and longer mesh face amount.

## D. More Results

We present additional qualitative comparisons with BPT and DeepMesh in Figure 3, where the face count for each mesh is explicitly annotated. Furthermore, extended results for point cloud-conditioned generation are illustrated in Figure 4 and Figure 5. Finally, Figure 6, Figure 7, Figure 8 and Figure 9 displays high-resolution visualizations to

Table 3. **GPU memory (GB) and runtime (s) comparison w/o NSCA on a single A800 GPU.** Each configuration is evaluated and averaged over 128 iterations. Context and window sizes are measured in number of faces.

Total Faces	Window	NSCA		w/o NSCA	
		Time	Mem	Time	Mem
5K	1K	0.253	24.65	0.180	27.68
5K	2K	0.391	29.94	0.335	33.68
5K	5K	1.199	47.85	1.156	53.59
10K	1K	0.252	24.72	0.199	28.50
10K	2K	0.393	30.01	0.362	35.17
10K	5K	1.194	47.91	1.207	57.13
20K	1K	0.254	24.86	0.244	30.16
20K	2K	0.398	30.16	0.419	38.15
20K	5K	1.200	48.05	OOM	
50K	1K	0.265	25.34	0.426	35.09
50K	2K	0.420	30.61	0.636	47.19
50K	5K	1.229	48.46	OOM	
100K	1K	0.300	26.13	0.726	43.35
100K	2K	0.458	31.42	0.984	62.26
100K	5K	1.275	49.19	OOM	

facilitate a detailed inspection of the geometric fine structures.





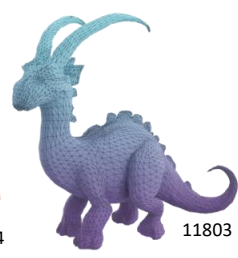


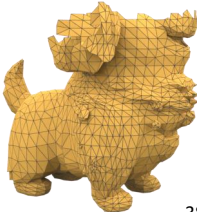
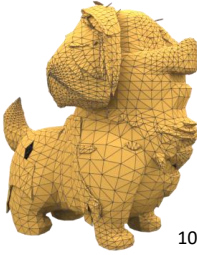








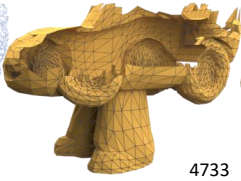












Origin Mesh	Point Cloud	BPT	DeepMesh	Ours
		 1915	 13454	 11803
		 3819	 10211	 12359
		 2898	 23152	 7550
		 4733	 18655	 13677
		 3740	 18568	 14171
		 2457	 6703	 7186

Figure 3. More comparisons between MeshRipple and the SOTAs.



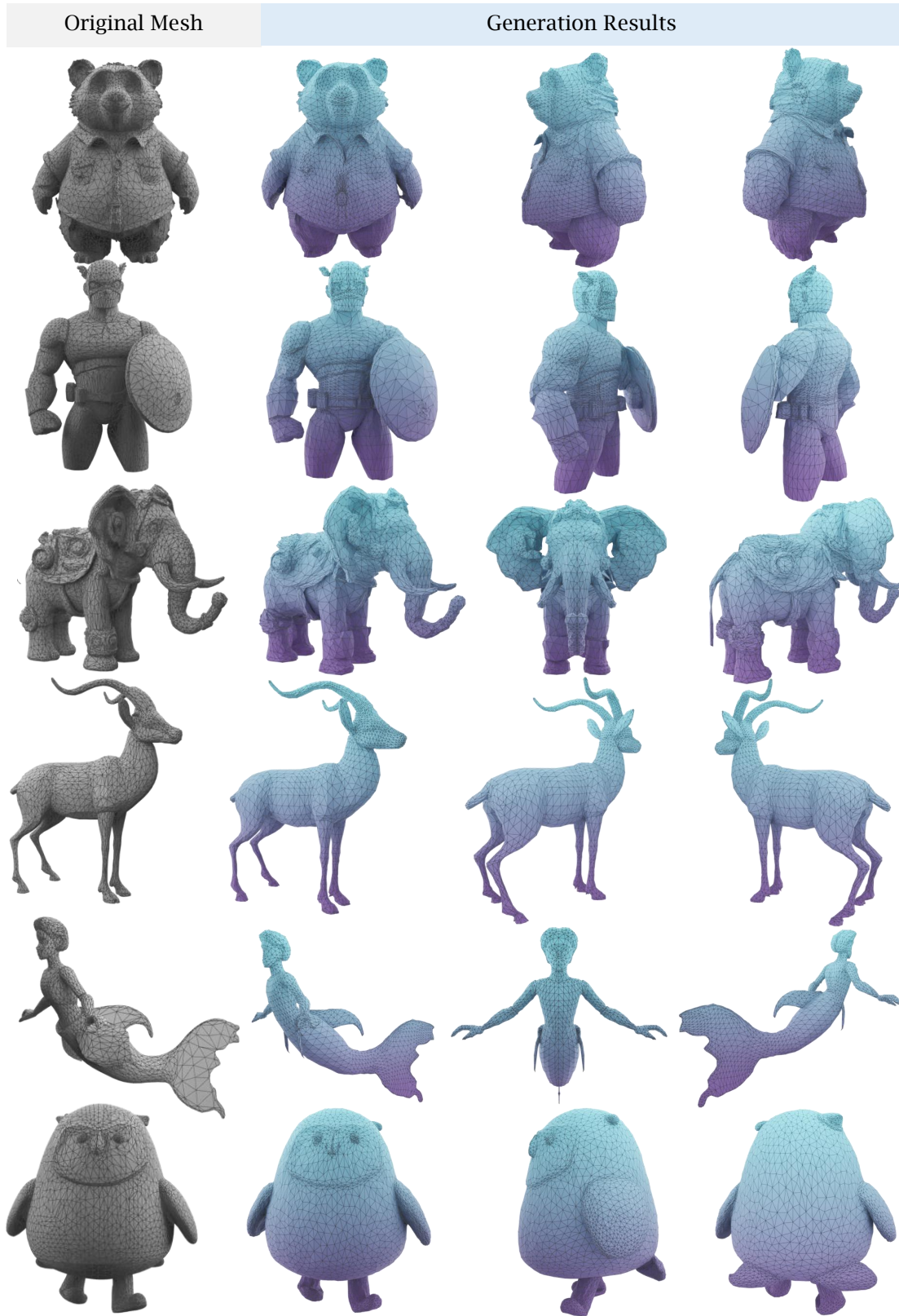


Figure 4. More results generated by MeshRipple.

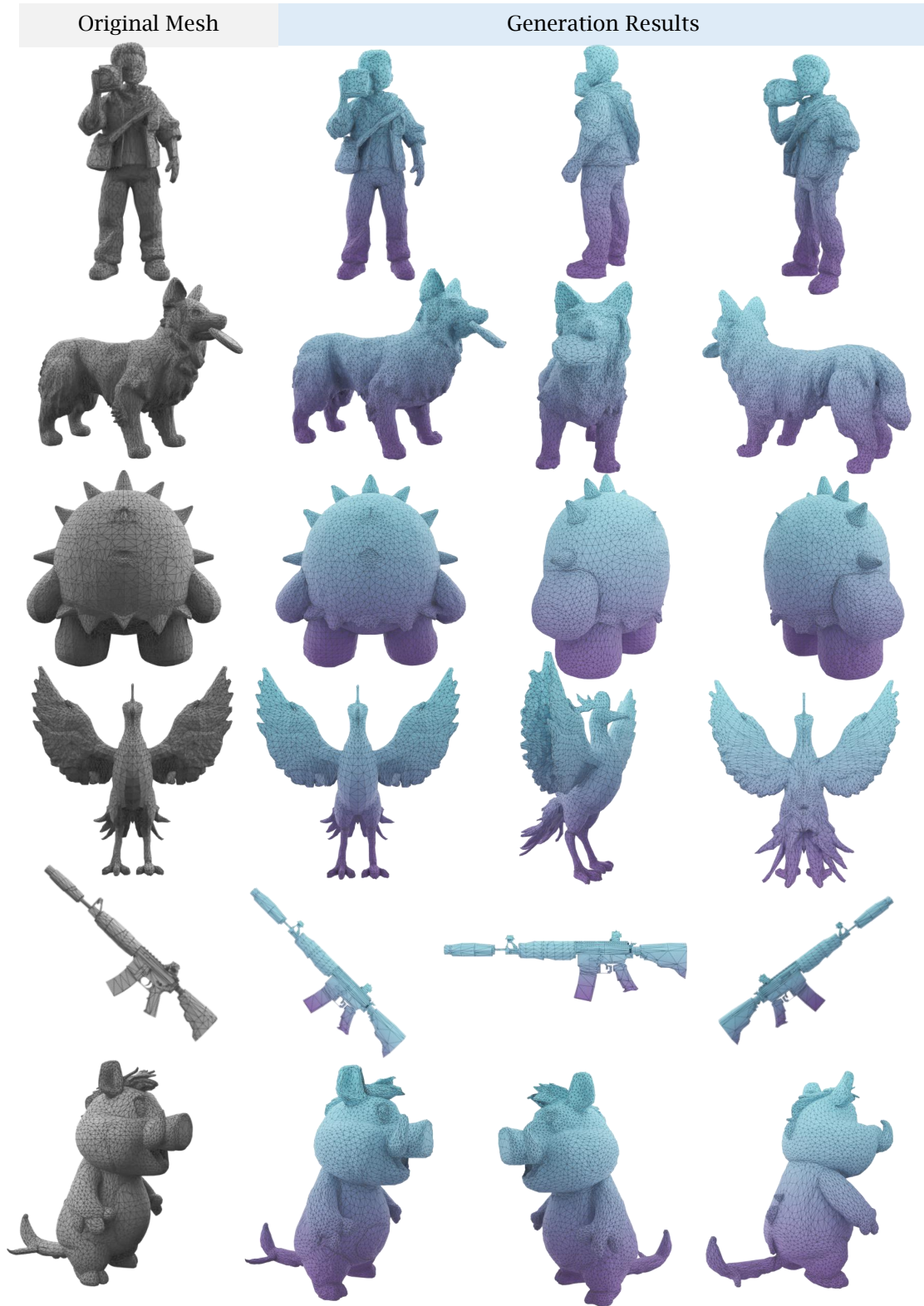


Figure 5. More results generated by MeshRipple.

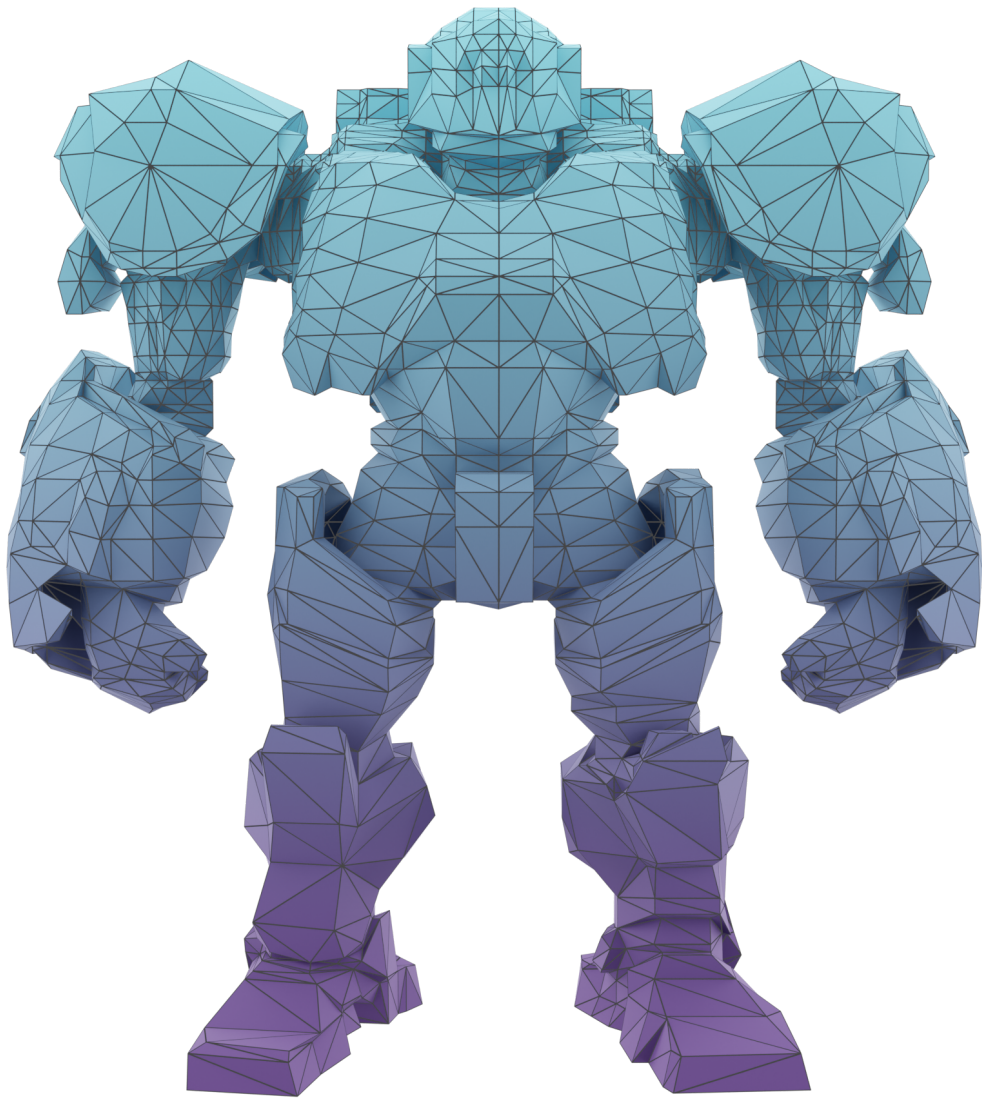


Figure 6. **High-fidelity visualizations of generated meshes.**



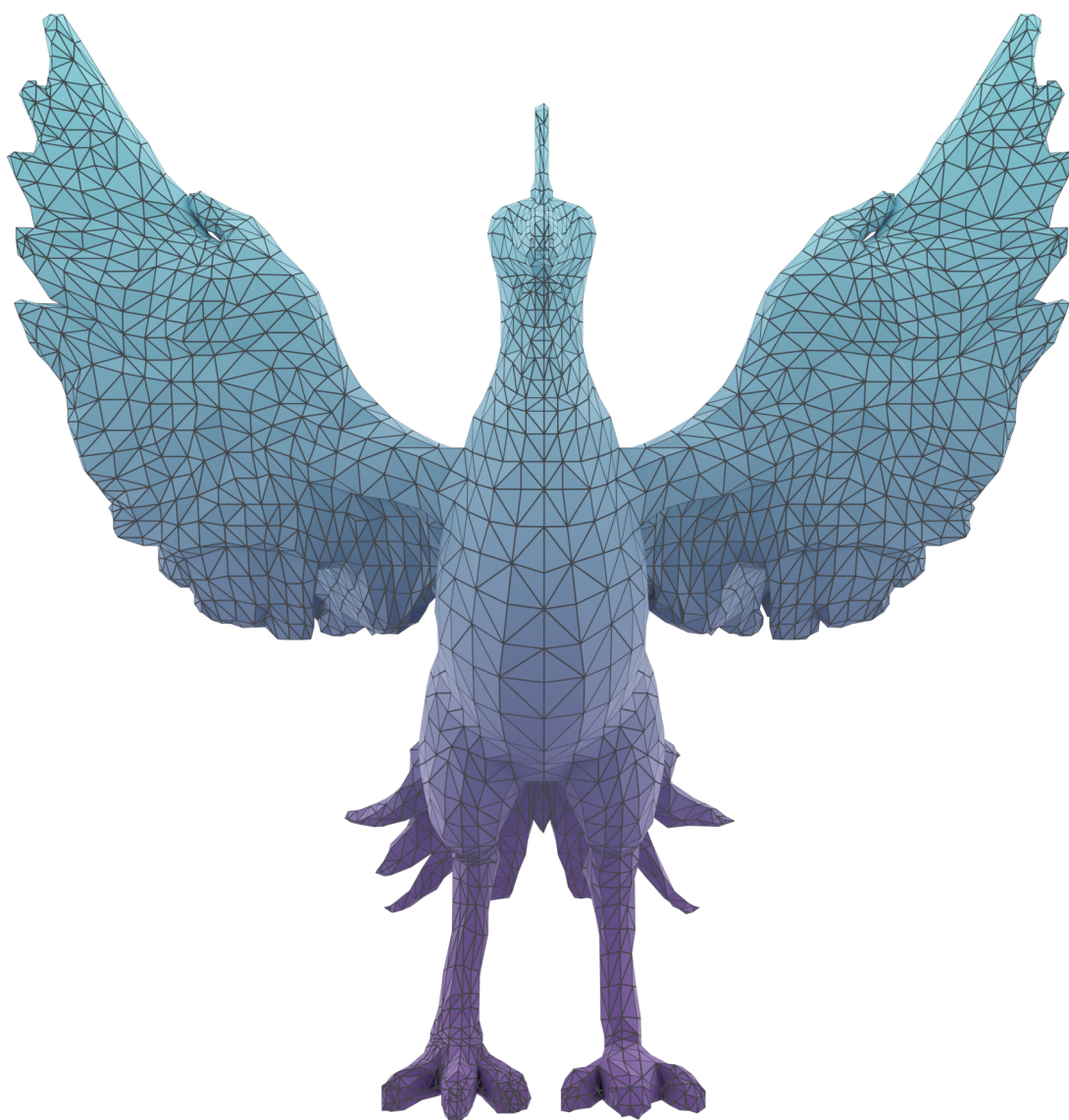


Figure 7. **High-fidelity visualizations of generated meshes.**

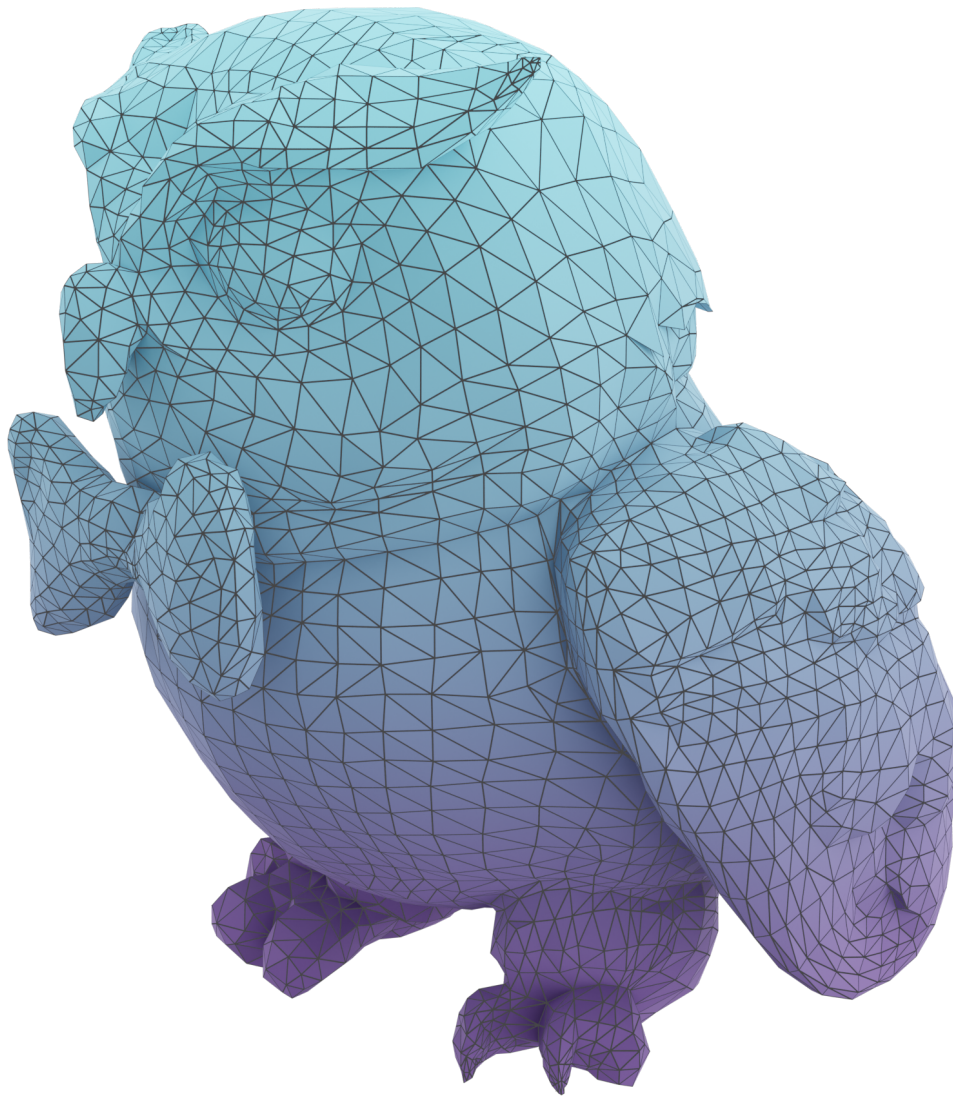


Figure 8. **High-fidelity visualizations of generated meshes.**



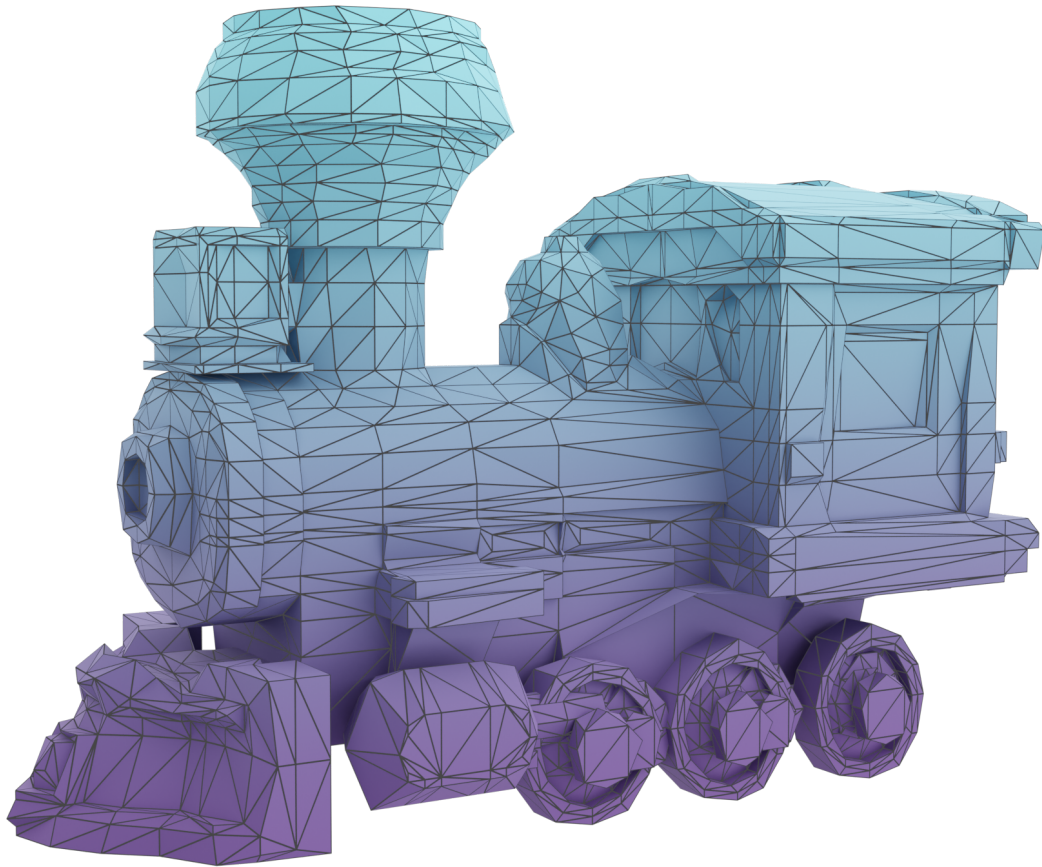


Figure 9. **High-fidelity visualizations of generated meshes.**