TRIEMESH: STRUCTURED FRONTIER REPRESENTATION FOR AUTOREGRESSIVE ARTISTIC MESH GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Autoregressive artistic mesh generation has progressed via compact tokenization, yet each prediction step still mainly relies on sequence history to represent the current mesh state. The active frontier is an important part of this state, but is rarely represented explicitly in a structured and queryable form. We present **TrieMesh**, a frontier-guided autoregressive representation that maintains active frontier edges in a Trie with canonical lexicographic keys, enabling deterministic selection of the next edge for expansion. A shared autoregressive decoder switches between full-face generation when the frontier is empty and completion-vertex prediction otherwise, reusing the two known endpoints of the active edge to avoid redundant coordinate prediction. Experiments show improved geometric fidelity over recent autoregressive baselines under comparable token budgets. Beyond global generation, the same structured frontier representation naturally supports mesh hole filling, where the boundary edges of a missing region initialize the active frontier. This allows global mesh generation and local mesh editing to follow the same frontier-guided autoregressive expansion process.

1 INTRODUCTION

Artistic mesh generation has attracted growing attention in recent years. Among existing approaches, autoregressive methods (Nash et al., 2020; Chen et al., 2024b;a; 2025; Tang et al., 2024; Weng et al., 2025; Lionar et al., 2025; Wang et al., 2026a) represent mesh geometry and topology as discrete tokens and serialize them into one-dimensional sequences. A model is then trained to predict the next token conditioned on the generated prefix, progressively constructing the mesh. Unlike diffusion or flow methods (Zhao et al., 2026; Long et al., 2026; Li et al., 2026a;b; Wang et al., 2026b), this incremental process maintains a partial mesh at every step, making the **frontier**, the set of open edges along which generation can continue, a natural structural state for subsequent growth.

Recently, TreeMeshGPT (Lionar et al., 2025) maintains active frontiers with a DFS stack, while MeshRipple (Lin et al., 2025) uses a BFS queue together with a learned routing head. These methods show that explicit frontier information provides useful geometric context for autoregressive expansion. However, frontier selection is still tied to a traversal policy or an additional routing decision, rather than a canonical frontier representation, making it difficult to treat the frontier as a stable structural state that can be directly queried and integrated into decoding.

Meanwhile, FACE (Wang et al., 2026a) shows that one-face-one-token representation with cascaded CausalMLP decoding can substantially shorten generation sequences. However, compact face-level tokenization alone neither determines which incomplete boundary should be expanded next nor exploits the two known vertices already provided by an active frontier. This raises a key question: *Can the evolving frontier be represented as a canonical, queryable state while preserving face-level efficiency and directly guiding face decoding?*

Therefore, we propose **TrieMesh**, in which each active frontier edge is assigned a canonical lexicographic key and stored in a Trie, allowing the next expansion edge to be selected deterministically. Following FACE, each triangle is encoded as one face token and decoded by a cascaded CausalMLP. On top of this representation, a dual-mode inference strategy switches between full-face generation when the frontier is empty and frontier-conditioned completion otherwise: the two known endpoints

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Table 1: Comparison on frontier representations. TrieMesh preserves the compact token efficiency of FACE (Wang et al., 2026a) while explicitly modeling the evolving frontier as an ordered and queryable structural state for mesh growth.

Capability	TreeMeshGPT	MeshRipple	FACE	TrieMesh (Ours)
Frontier Management	✓	✓	×	✓
Frontier Representation	LIFO Stack	FIFO Queue	–	Trie
Next Expansion	LIFO	Learned	–	Lexicographic
Canonical Ordering	×	×	–	✓
Frontier Query	$O(n)$	$O(n)$	–	$O(d)$
Frontier Deletion	$O(n)$	$O(n)$	–	$O(d)$
Compression Ratio ↓	0.22	1.0	0.11	≈ 0.12

are reused as context, and only the new vertex is predicted. Residual frontier edges induced by open boundaries and odd non-manifold edges are terminated by lightweight virtual faces. As summarized in Tab. 1, TrieMesh provides a structured, queryable frontier state while retaining a compression ratio comparable to FACE. The same representation naturally extends to mesh hole filling, where the boundary of a missing region directly initializes the active frontier and guides local patch completion. This allows global mesh generation and local mesh editing to follow the same frontier-guided autoregressive expansion process.

In summary, our contributions are as follows:

- We introduce a frontier-guided autoregressive representation for mesh generation with face-level tokenization, in which a Trie maintains a canonical lexicographic frontier state for deterministic expansion.
- We introduce a dual-mode inference strategy that generates a full face when the frontier is empty and otherwise reuses the two frontier vertices to predict only the new vertex, thereby avoiding inconsistencies caused by re-predicting known vertices.
- We show that our frontier-guided representation unifies global mesh generation and local editing through a shared autoregressive expansion process. Experiments demonstrate high geometric fidelity and mesh quality across both tasks.

2 Related Work

2.1 Artistic Mesh Generation

One line of research formulates artistic mesh generation as autoregressive sequence modeling, representing mesh geometry and topology as 1D sequences. Early two-stage approaches such as PolyGen (Nash et al., 2020) first generate vertices and then predict face connectivity. MeshGPT (Siddiqui et al., 2024) introduced a single-stage, end-to-end paradigm, later adopted by MeshXL (Chen et al., 2024a) and PivotMesh (Weng et al., 2024). However, the resulting long sequences limit generation efficiency, motivating various sequence compression strategies. EdgeRunner (Tang et al., 2024), MeshAnythingV2 (Chen et al., 2025), and MeshSilksong (Song et al., 2026) exploit mesh adjacency and topology-aware traversal to improve vertex reuse. TreeMeshGPT (Lionar et al., 2025), BPT (Weng et al., 2025), Nautilus (Wang et al., 2025b), and FastMesh (Kim et al., 2026) explore compact tokenization schemes, while Meshtron (Hao et al., 2024) and iFlame (Wang et al., 2025a) improve efficiency through architectural designs. Beyond efficiency, DeepMesh (Zhao et al., 2025) improves generation quality through reinforcement learning, while MeshMosaic (Xu et al., 2026) uses local-to-global assembly for large-scale mesh generation.

More recently, another line of research has explored flow- and diffusion-based approaches to model mesh geometry and connectivity in latent spaces. Methods such as LATO (Zhao et al., 2026), LATO.2 (Long et al., 2026), MeshFlow (Li et al., 2026b), TriFlow (Li et al., 2026a), and Nexus (Wang et al., 2026b) avoid autoregressive token-by-token decoding, enabling faster generation and providing a promising alternative for large-scale mesh generation.

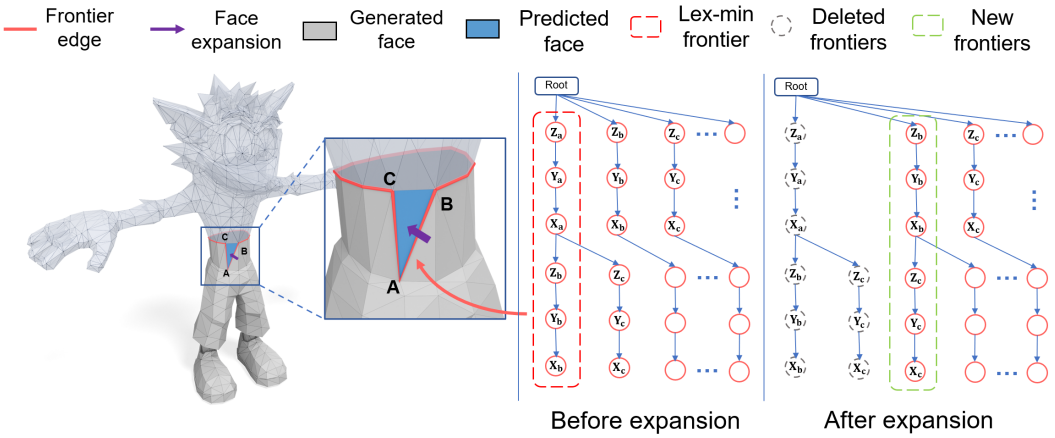

Figure 1: Illustration of the Trie-based frontier expansion and update process. The lexicographically smallest frontier edge AB is selected and expanded into triangle ABC on the mesh (left). The Trie states before (middle) and after (right) expansion show that AB is consumed. Of the two candidate edges, AC already exists and is removed as a closed frontier edge, whereas BC is absent and is inserted as a new frontier edge.

2.2 Explicit Frontier Representation

Beyond sequence efficiency and non-autoregressive alternatives, we focus on an important challenge in autoregressive mesh generation is how to explicitly represent and maintain the current frontier. Recent applications such as MeshFIM (Yang et al., 2026) also show the value of explicit boundary information, which helps local mesh generation stay consistent with the surrounding geometry. TreeMeshGPT (Lionar et al., 2025) maintains the frontier with a DFS stack, where mesh expansion follows a LIFO order and inherently requires backtracking, resulting in longer sequences. MeshRipple (Lin et al., 2025) adopts a BFS queue instead, together with an additional prediction head for selecting the next expansion root. These methods demonstrate the benefit of explicitly maintaining the current frontier state, but the selection of the next frontier edge to expand remains tightly coupled to the traversal order. In contrast, TrieMesh stores the frontier as a canonical and queryable state, where selection is determined by the lexicographical order of the frontier edges, enabling deterministic expansion while maintaining a low compression ratio comparable to existing baselines.

3 Proposed Method

3.1 Preliminaries: Trie Data Structure

A Trie is a tree-structured data structure for storing discrete keys, supporting insertion, lookup, and deletion in $O(d)$ time for keys of length d. In contrast, locating a specific edge in a stack or queue requires $O(n)$ search over n frontier edges (Tab. 1). In TrieMesh, each quantized frontier edge is canonically encoded by the discrete 3D coordinates of its two endpoints, forming a length-6 key stored in a depth-6 Trie (Fig. 1). This enables efficient insertion, query, and deletion of frontier edges during mesh growth. Further details are provided in Appendix A.

3.2 Trie-based Structured Frontier Representation

Each frontier edge is formed by two endpoint vertices, with $v_i = (z_i, y_i, x_i)$ denoting the quantized vertex coordinates. For a frontier edge (v_i, v_j), we impose a lexicographic order such that $v_i < v_j$, and define its canonical key as

$$k(v_i, v_j) = (z_i, y_i, x_i, z_j, y_j, x_j). \tag{1}$$

These canonical frontier keys are stored in a Trie, where each frontier edge is represented by a unique root-to-leaf path. This allows the active frontier edges to be maintained in lexicographic order and accessed directly by their keys.

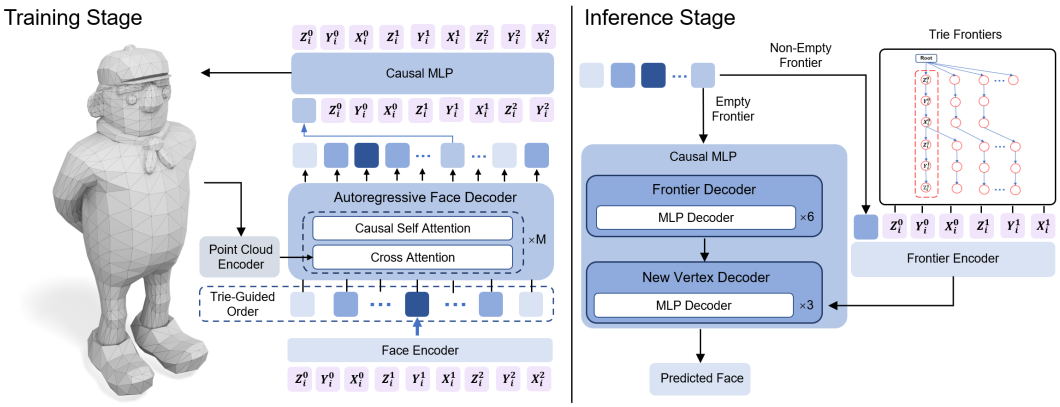

Figure 2: Overview of TrieMesh. Left: training architecture with Trie-guided face ordering. Right: dual-mode inference controlled by the Trie state. An empty Trie triggers full-face generation, while a non-empty Trie provides a frontier edge for completion-vertex prediction.

At each expansion step, the lexicographically smallest frontier edge is removed from the Trie and expanded into a new face. The other two edges are then queried: an existing frontier edge is removed, while a new one is inserted, as illustrated in Fig. 1.

Unlike TreeMeshGPT (Lionar et al., 2025) and MeshRipple (Lin et al., 2025), which rely on a traversal policy and a learned routing head, respectively, TrieMesh determines the expansion order directly from canonical frontier keys. As a result, frontier expansion depends only on the current frontier state, rather than the insertion order, and requires no additional routing head.

3.3 Frontier-guided Autoregressive Generation

Trie-Guided Face Ordering. FACE (Wang et al., 2026a) shows that a simple, deterministic ZYX lexicographic face order performs well in face-level autoregressive generation, outperforming the traversal-based orders evaluated in their study. Motivated by this result, we apply lexicographic selection to active frontier edges, expanding the smallest edge at each step to determine the next face. Following FACE, we sort all triangles according to the lexicographic ZYX order (Chen et al., 2024a; Siddiqui et al., 2024) and select the lexicographically smallest triangle as the seed face. We initialize a Trie with the three edges of the seed face, and then expand the frontier according to triangle adjacency following the procedure described in Sec. 3.2. Once the Trie becomes empty, we select the lexicographically smallest triangle among the remaining unvisited faces as a new seed and reinitialize the Trie. The process continues until all triangles have been visited.

Model Architecture. As shown in the left panel of Fig. 2, TrieMesh comprises a point-cloud encoder and an autoregressive face decoder. We use Michelangelo (Zhao et al., 2023) to encode the input point cloud into a compact latent representation C. Following FACE (Wang et al., 2026a), the face decoder consists of a face encoder, a multi-layer Transformer decoder, and a cascaded CausalMLP head.

The face encoder maps the previously generated face f_{i-1} into a face token:

$$t_{i-1} = \mathrm{MLP}_{\mathrm{embed}}(f_{i-1}). \tag{2}$$

The Transformer decoder then processes the face tokens through multiple layers, each containing causal self-attention and cross-attention with the point-cloud latent C:

$$\begin{aligned} H'_l &= \mathrm{CausalSelfAttn}(H_l), \\ H_{l+1} &= \mathrm{CrossAttn}(Q = H'_l,\ K = C,\ V = C). \end{aligned} \tag{3}$$

Finally, the predicted face token is decoded by a 9-layer cascaded CausalMLP (Lionar et al., 2025) into the 9 quantized coordinate tokens of the three vertices. Specifically, the CausalMLP performs a second level of autoregressive decoding within each face: the i-th coordinate token is predicted conditioned on the face latent and all previously decoded coordinate tokens of the same face. The cascaded design also makes it possible to reuse known frontier information during decoding, leading to the dual-mode inference strategy described next.

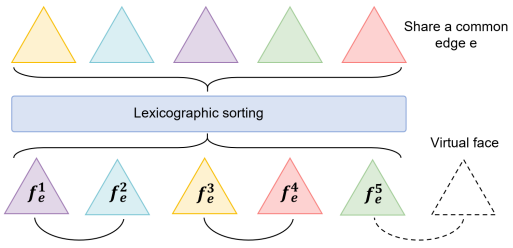

Figure 3: Illustration of non-manifold face pairing. Faces sharing the same edge e are sorted in lexicographic order and paired consecutively. For an odd number of incident faces, the last unpaired face is matched with a virtual face.

Figure 4: Statistics of virtual-face overhead. Per-mesh overhead is normalized by the number of real faces F. Dataset-level overhead reports the overall additional token ratio, while zero-overhead meshes denote the fraction requiring no virtual faces.

Metric	Mean (%)	Median (%)
Boundary overhead	9.57	5.80
Odd non-manifold	5.14	2.53
Total overhead	**14.71**	**11.11**
Dataset-level overhead	13.74%	
Zero-overhead meshes	13.34%	

Dual-mode Inference. As illustrated by the inference pipeline in Fig. 2, we divide the 9-layer cascaded CausalMLP into two stages: a 6-layer *frontier decoder* for the first two vertices and a 3-layer *new-vertex decoder* for the third vertex. This design allows the first stage to be skipped when the frontier vertices are already available. TrieMesh then switches between full-face generation and frontier-conditioned vertex generation according to the current Trie state $\mathcal{T}_i$:

$$p_i = \begin{cases} p(f_i \mid f_{<i}, C), & \mathcal{T}_i = \varnothing, \\ p(v_2 \mid v_0, v_1, f_{<i}, C), & \mathcal{T}_i \neq \varnothing,\ (v_0, v_1) = \min_{\text{lex}}(\mathcal{T}_i). \end{cases} \tag{4}$$

For an empty Trie, $\mathcal{T}_i = \varnothing$, no active frontier is available. TrieMesh therefore performs full-face generation and uses the complete 9-layer CausalMLP to predict all three vertices of f_i. The generated face is then used to initialize a new frontier Trie.

For a non-empty Trie, $\mathcal{T}_i \neq \varnothing$, TrieMesh selects the lexicographically smallest frontier (v_0, v_1). Since the first two vertices are already known, their six coordinate tokens are directly used as the prefix of the face, allowing the first six CausalMLP layers to be skipped. The model only applies the last three layers to predict the new vertex v_2, forming the next face $f_i = (v_0, v_1, v_2)$.

This dual-mode design reuses the explicit frontier information whenever available, reducing coordinate-level decoding from nine stages to three for most frontier-guided generation steps. By treating the two frontier vertices as known context, the decoder predicts only the completion vertex. This avoids re-predicting vertices whose coordinates have already been fixed, thereby preventing potential inconsistencies introduced by repeated coordinate predictions.

3.4 Residual Frontiers and Compression Analysis

Virtual Faces. The frontier expansion process described above naturally closes an edge when it is shared by two generated faces. However, open boundaries and non-manifold structures may leave unmatched frontier edges after all real faces have been generated. Let d_e denote the number of real faces incident to an edge e. An unmatched frontier edge remains when

$$d_e = \begin{cases} 1, & \text{Open boundary edge}, \\ > 2 \text{ and odd}, & \text{Odd non-manifold edge}. \end{cases} \tag{5}$$

For these cases, we introduce a *virtual face* to consume the remaining frontier edge without inserting any new frontier edge into the Trie. In contrast, non-manifold edges with an even number of incident faces are fully paired and require no virtual face. Virtual faces are used only to terminate unmatched frontiers during autoregressive generation and are removed from the final mesh.

Pair-aware Face Adjacency. We construct a pair-aware traversal adjacency for every mesh edge. Boundary edges are paired with a virtual face, while manifold edges have a unique pair of incident faces. For a non-manifold edge e with $d_e > 2$, we sort its incident faces in lexicographic ZYX order (Chen et al., 2024a; Siddiqui et al., 2024), as shown in Fig. 3:

$$\pi_e = (f_e^1, f_e^2, \ldots, f_e^{d_e}), \qquad f_e^k <_{\text{lex}} f_e^{k+1}, \tag{6}$$

and pair consecutive faces as

$$(f_e^1, f_e^2),\ (f_e^3, f_e^4), \dots. \tag{7}$$

If d_e is odd, the final face is paired with a virtual face. Each pair is treated as a separate traversal edge associated with e. The pairing specifies which two faces are connected across this edge, while the Trie traversal determines their positions in the sequence. The original mesh itself is not modified.

Compression Analysis. The virtual faces introduced above slightly increase the sequence length compared with the standard one-face-one-token representation (Wang et al., 2026a). Let F denote the number of real faces, B the number of open boundary edges, and N_{odd} the number of non-manifold edges with an odd number of incident faces. Since each unmatched frontier edge requires one virtual face, the total number of face tokens becomes

$$N_{\text{token}} = F + \underbrace{B + N_{\text{odd}}}_{\text{additional overhead}}. \tag{8}$$

To quantify the practical overhead in a reproducible setting, we further evaluate on the public ShapeNet subset used by MeshGPT (Siddiqui et al., 2024), containing approximately 28K meshes. Its public availability allows us to report overhead statistics independently of our own data curation procedure. As shown in Fig. 4, the average per-mesh overhead is 14.71%, with a median of 11.11%, while the overall dataset-level overhead is 13.74%. In addition, 13.34% of the meshes require no virtual faces. These results show that TrieMesh maintains a compression ratio of approximately 0.12, remaining close to the one-face-one-token baseline.

4 Experiments

4.1 Implementation Details

Training Data. We construct the training set from Objaverse (Deitke et al., 2023) and ShapeNet (Chang et al., 2015), retaining meshes with fewer than 5,000 faces to keep the autoregressive sequences within a practical range. We further filter out meshes whose virtual-face overhead exceeds 20% of the original face count, since excessive residual frontiers would introduce a large number of additional virtual-face tokens. After filtering, approximately 100K meshes remain for training. This procedure removes meshes with unusually high virtual-face overhead while still preserving samples containing moderate open boundaries and non-manifold structures.

Model and Optimization. We use a 24-layer decoder-only Transformer with a hidden dimension of 1024, comprising approximately 500M trainable parameters. The model is trained for 100K steps on 8 NVIDIA A100 80GB GPUs using AdamW with a learning rate of 10^{-4}, $\beta_1 = 0.9$, $\beta_2 = 0.99$, and a weight decay of 0.1.

Conditioning and Data Augmentation. For each mesh, we uniformly sample 8,192 surface points with normals as input to the point encoder. For data augmentation, we apply anisotropic scaling with independent per-axis factors sampled from $\mathcal{U}(0.75, 0.9)$, followed by a random rotation around the y-axis, with the rotation angle sampled uniformly from $[0°, 360°)$.

Loss. We use 7-bit quantization for vertex coordinates, with each coordinate quantized to an integer in the range $[0, 127]$. A real face is represented as $(z_0, y_0, x_0, z_1, y_1, x_1, z_2, y_2, x_2)$, a virtual face as $(z_0, y_0, x_0, z_1, y_1, x_1, \text{EOS}, \text{PAD}, \text{PAD})$, and sequence termination as $(\text{EOS}, \text{PAD}, \dots, \text{PAD})$. The EOS token is shared across slots, with its semantics determined by its slots. The training objective is the masked cross-entropy loss over all valid tokens:

$$\mathcal{L} = \frac{1}{\sum_{i,j} m_{i,j}} \sum_{i=1}^{N} \sum_{j=1}^{9} m_{i,j}\, \text{CrossEntropy}(L_{i,j}, c_{i,j}), \tag{9}$$

where $m_{i,j}$ masks out padded tokens.

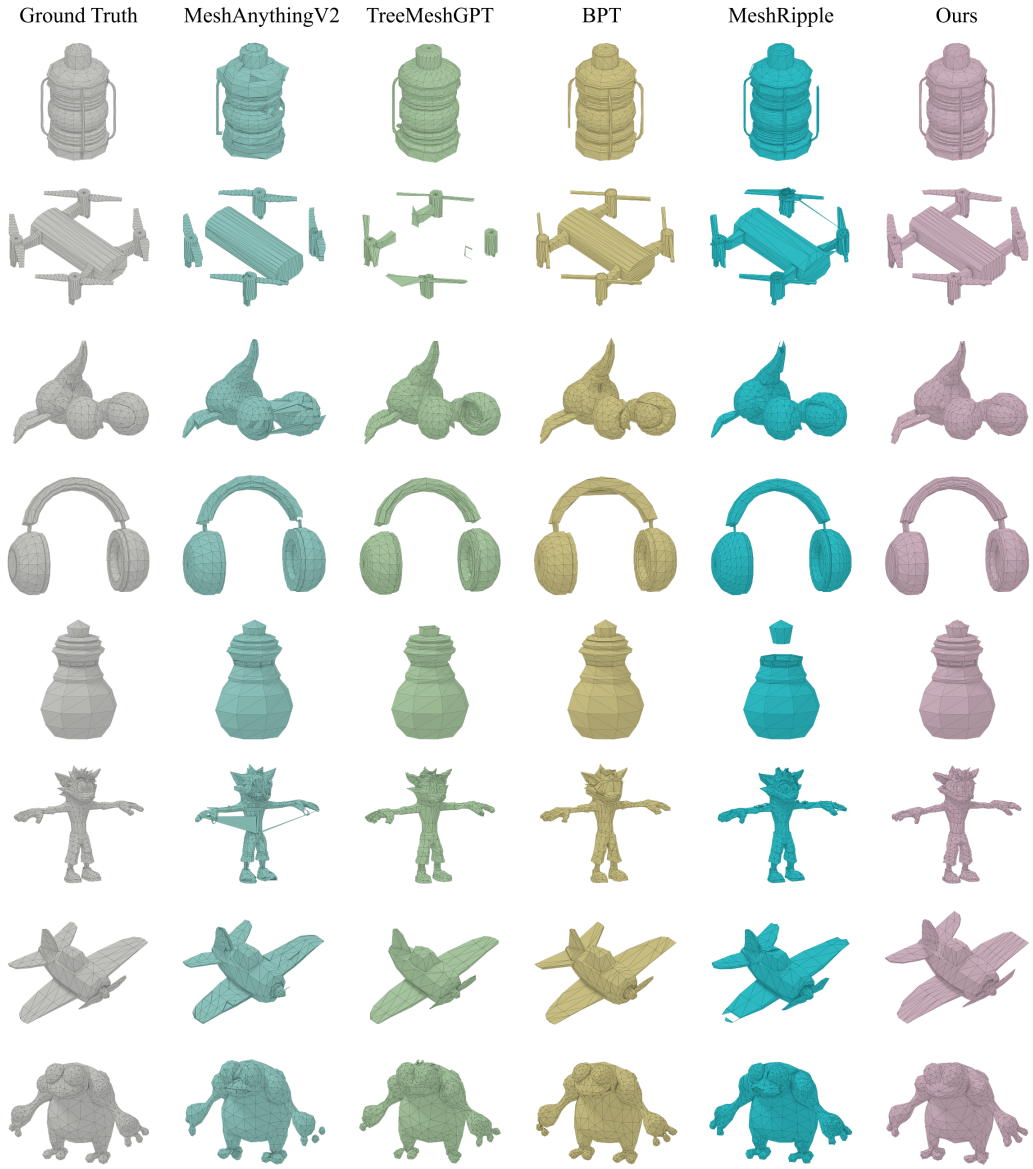

Figure 5: Qualitative comparison of point-cloud-conditioned mesh generation. Across the examples shown, TrieMesh closely follows the ground-truth geometry and retains thin or small components that are missing or disconnected in some baseline results.

4.2 Comparison with Baselines

We compare TrieMesh with MeshAnything V2 (Chen et al., 2025), TreeMeshGPT (Lionar et al., 2025), BPT (Weng et al., 2025), and MeshRipple (Lin et al., 2025) under 7-bit coordinate quantization, using their released pretrained checkpoints. As shown in Fig. 5, TrieMesh closely follows the reference shapes while preserving thin structures and small components. In several baseline results, parts such as the drone arms are missing or disconnected, and local components are generated separately from the main mesh. For quantitative evaluation, we use 300 Objaverse meshes disjoint from the training set and 200 Toys4K (Stojanov et al., 2021) meshes, all with fewer than 5,000 faces. In Tab. 2, TrieMesh achieves the lowest Hausdorff and Chamfer distances on both datasets: $0.094/0.036$ on Objaverse and $0.064/0.027$ on Toys4K, respectively. Overall, the numerical results are consistent with the qualitative examples, suggesting that TrieMesh generally preserves the overall shape while retaining finer local structures.

Table 2: Quantitative comparison of mesh generation quality on Objaverse and Toys4K.

Method	Objaverse		Toys4K	
	Hausdorff ↓	Chamfer ↓	Hausdorff ↓	Chamfer ↓
MeshAnything V2	0.228	0.097	0.193	0.084
TreeMeshGPT	0.177	0.059	0.128	0.051
BPT	0.116	0.042	0.078	0.032
MeshRipple	0.127	0.047	0.086	0.039
Ours	**0.094**	**0.036**	**0.064**	**0.027**

Table 3: Ablation study of frontier-based face ordering and dual-mode inference. Full 9-slot denotes full-face decoding, whereas dual-mode uses 9-slot decoding for seed faces and 3-slot decoding for expansion faces.

Setting	Hausdorff ↓	Chamfer ↓
ZYX	0.124	0.061
Ours (Full 9-slot)	0.129	0.066
Ours (Dual-mode)	**0.112**	**0.053**

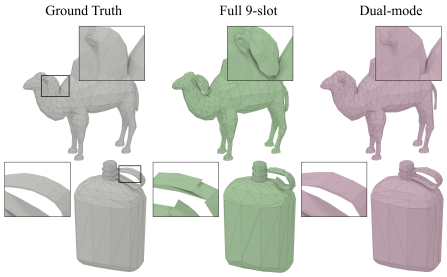

Figure 6: Representative examples of full 9-slot and dual-mode decoding.

4.3 Ablation Study

We conduct three ablation studies on face ordering and dual-mode inference, supervision for expansion faces, and residual-frontier handling. All experiments use a decoder with a hidden dimension of 768.

Ablation on face ordering and dual-mode inference. Tab. 3 compares a FACE-style ZYX baseline (Wang et al., 2026a), our Trie-guided ordering with full 9-slot decoding, and the same ordering with Trie-assisted dual-mode inference. The Trie-guided face ordering alone performs comparably to ZYX, with HD/CD of 0.129/0.066 versus 0.124/0.061. Adding Trie-assisted dual-mode inference reduces the errors to 0.112/0.053, suggesting that the main gain comes from using frontier information during decoding. The examples in Fig. 6 show corresponding local differences. Around the camel's neck, dual-mode produces a more continuous connection where full 9-slot decoding leaves a visible gap. On the kettle's thin handle, it also avoids the slightly misaligned surface strips seen in the full 9-slot result.

Supervision for Expansion Faces. During expansion, the Trie provides the two frontier vertices, so supervising only the completion vertex may appear sufficient. However, this setting substantially degrades generation quality: HD and CD increase from 0.112/0.053 to 0.553/0.338 (Tab. 4). Expansion faces constitute most of the training sequence. Without supervision on their first six coordinate slots, the corresponding prediction stages are trained mainly on the relatively sparse seed faces. This weakens full-face decoding when the Trie is empty; an inaccurate seed face can also initialize incorrect frontiers for subsequent expansion. We therefore supervise all nine coordinates for both seed and expansion faces during training, while reusing the known frontier vertices at inference.

Strategies for Handling Residual Frontiers. We compare virtual faces with two learned-routing variants that predict a 3-bit token identifying edges to exclude from the Trie. Parallel routing uses a parallel head, while cascaded routing adds a tenth decoding stage; details are given in Appendix D. Both variants retain a compression ratio of 0.11, compared with approximately 0.12 for virtual faces, but yield substantially higher CD and HD (Tab. 5). The routing-token distribution is highly imbalanced, likely making rare routing configurations difficult to predict. Such errors can alter the Trie state during subsequent expansion. Virtual faces instead treat frontier termination as the completion of a face, avoiding an additional learned routing decision and achieving better geometric quality at a small sequence cost.

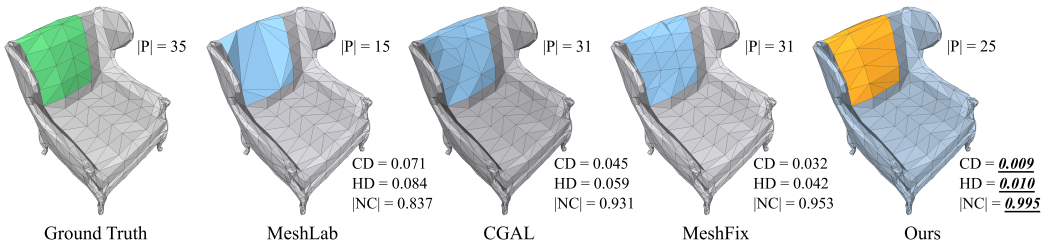

Figure 7: Application to mesh hole filling. Colored regions mark the hole-filling area, and the removed ground-truth patch contains 35 faces. Our method reconstructs the region using 25 faces, compared with 35 in the ground-truth patch, while achieving the lowest CD and HD among the compared methods and producing a triangulation pattern better aligned with common artist modeling preferences.

Table 4: Ablation of coordinate supervision on expansion faces. Full 9-slot CE yields lower CD and HD than supervising only the completion vertex, although frontier vertices are available at inference.

Loss Setting	Hausdorff ↓	Chamfer ↓
w/o frontier CE	0.553	0.338
Full 9-slot CE	**0.112**	**0.053**

Table 5: Ablation of residual-frontier handling strategies. The virtual-face strategy achieves lower CD and HD at the cost of a slightly higher compression ratio.

Metric	Parallel Routing	Cascaded Routing	Virtual Faces
Compression Ratio ↓	0.11	0.11	≈ 0.12
Hausdorff ↓	0.313	0.248	**0.112**
Chamfer ↓	0.192	0.147	**0.053**

5 Application to Mesh Hole Filling

We further apply the proposed Trie-based structured frontier representation to mesh hole filling. Given an incomplete triangle mesh, the boundary of the missing region naturally forms a set of active frontiers, which can be directly organized and updated using the Trie. Starting from these boundary edges, the missing patch is autoregressively generated by expanding the current frontier until all frontiers are closed.

As shown in Fig. 7, we compare against three widely used hole-filling implementations on the same incomplete mesh: the hole-closing operator in MeshLab (Cignoni et al., 2008), and the polygon-mesh hole-filling pipeline in CGAL (The CGAL Project, 2026) and MeshFix (Attene, 2010). For the repaired region, we report CD, HD, absolute normal consistency ($|NC|$), and the number of generated patch faces $|P|$, jointly measuring reconstruction fidelity and patch complexity. By maintaining the evolving hole boundary as a structured frontier representation, our method achieves low geometric errors, strong normal agreement, and structured triangulations aligned with common artist modeling preferences. These results demonstrate the value of the proposed representation for compact and accurate mesh repair. Additional implementation details and results are provided in Sec. F.

6 Conclusion

We presented **TrieMesh**, an autoregressive representation that maintains the evolving mesh frontier as an explicit structured state. Canonical Trie keys enable deterministic frontier expansion and updates, while dual-mode inference reuses known frontier vertices and predicts only the completion vertex. Experiments demonstrate improved geometric fidelity while preserving a compact autoregressive representation. The same frontier-guided process also supports mesh hole filling, providing a unified representation for global generation and local editing.

Beyond these tasks, our structured frontier representation could provide an interface for controllable mesh generation, allowing users or external editing agents to select where the mesh should expand next. More broadly, TrieMesh highlights the potential of explicit data structures to connect neural generation with evolving geometric topology.

AI USE STATEMENT

In this work, we used generative AI tools for language editing and polishing of the manuscript. We have not used generative AI tools for developing research ideas, designing the methodology, conducting experiments, analyzing results, or formulating scientific claims. We reviewed all AI-assisted edits to ensure that they preserve the intended technical meaning and accurately reflect our work. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The main paper describes the Trie-based frontier representation, dual-mode inference, virtual-face handling, model architecture, and experimental protocol. Additional training settings, data processing, and point-cloud sampling are provided in Appendix B. Complete serialization pseudocode and complexity analysis are given in Appendix C. We further document the residual-frontier baselines in Appendix D, as well as the training, inference, baseline, and evaluation settings for mesh hole filling in Appendix F. The supplementary material also includes a video demonstrating the interactive mesh completion workflow.

REFERENCES

Marco Attene. A lightweight approach to repairing digitized polygon meshes. *The visual computer*, 26(11):1393–1406, 2010. 9, 22

Angel X Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, et al. Shapenet: An information-rich 3d model repository. *arXiv preprint arXiv:1512.03012*, 2015. 6

Sijin Chen, Xin Chen, Anqi Pang, Xianfang Zeng, Wei Cheng, Yijun Fu, Fukun Yin, Zhibin Wang, Jingyi Yu, Gang Yu, et al. Meshxl: Neural coordinate field for generative 3d foundation models. *Advances in Neural Information Processing Systems*, 37:97141–97166, 2024a. 1, 2, 4, 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*, 2024b. 1, 16

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. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 13922–13931, 2025. 1, 2, 7

Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. Meshlab: an open-source mesh processing tool. In *Eurographics Italian Chapter Conference*, pp. 129–136. The Eurographics Association, 2008. doi: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136. 9, 22

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*, pp. 13142–13153, 2023. 6

Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960. 13

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. 2

Jeonghwan Kim, Yushi Lan, Armando Fortes, Yongwei Chen, and Xingang Pan. Fastmesh: Efficient artistic mesh generation via component decoupling. In *2026 International Conference on 3D Vision (3DV)*, pp. 1478–1488. IEEE, 2026. 2

Haoxuan Li, Ziya Erkoç, Daniele Sirigatti, Vladislav Rosov, Lei Li, Angela Dai, and Matthias Nießner. Triflow: Generating artist-like 3d mesh topology via nearest-vertex vector fields. *arXiv preprint arXiv:2606.20131*, 2026a. 1, 2

Weiyu Li, Antoine Toisoul, Tom Monnier, Roman Shapovalov, Rakesh Ranjan, Ping Tan, and Andrea Vedaldi. Meshflow: Efficient artistic mesh generation via meshvae and flow-based diffusion transformer. *arXiv preprint arXiv:2606.04621*, 2026b. 1, 2

Junkai Lin, Hang Long, Huipeng Guo, Jielei Zhang, JiaYi Yang, Tianle Guo, Yang Yang, Jianwen Li, Wenxiao Zhang, Matthias Nießner, et al. Meshripple: Structured autoregressive generation of artist-meshes. *arXiv preprint arXiv:2512.07514*, 2025. 1, 3, 4, 7

Stefan Lionar, Jiabin Liang, and Gim Hee Lee. Treemeshgpt: Artistic mesh generation with autoregressive tree sequencing. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 26608–26617, 2025. 1, 2, 3, 4, 7

Hang Long, Tianhao Zhao, Junkai Lin, Youjia Zhang, Huipeng Guo, Rendong Liang, Jiale Xu, Jozef Hladkỳ, Matthias Nießner, and Wei Yang. Lato. 2: Factorized 3d mesh generation with vertex and topology flow. *arXiv preprint arXiv:2607.10623*, 2026. 1, 2

Charlie Nash, Yaroslav Ganin, SM Ali Eslami, and Peter Battaglia. Polygen: An autoregressive generative model of 3d meshes. In *International conference on machine learning*, pp. 7220–7229. PMLR, 2020. 1, 2

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*, pp. 19615–19625, 2024. 2, 4, 5, 6, 16, 20

Gaochao Song, Zibo Zhao, Haohan Weng, Jingbo Zeng, Rongfei Jia, and Shenghua Gao. Topology-preserved auto-regressive mesh generation in the manner of weaving silk. In *The Fourteenth International Conference on Learning Representations*, 2026. 2, 15

Stefan Stojanov, Anh Thai, and James M Rehg. Using shape to categorize: Low-shot learning with an explicit shape bias. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1798–1808. IEEE, 2021. 7

Jiaxiang Tang, Zhaoshuo Li, Zekun Hao, Xian Liu, Gang Zeng, Ming-Yu Liu, and Qinsheng Zhang. Edgerunner: Auto-regressive auto-encoder for artistic mesh generation. *arXiv preprint arXiv:2409.18114*, 2024. 1, 2

The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 6.2 edition, 2026. 9, 22

Hanxiao Wang, Biao Zhang, Weize Quan, Dong-Ming Yan, and Peter Wonka. iflame: Interleaving full and linear attention for efficient mesh generation. *arXiv preprint arXiv:2503.16653*, 2025a. 2

Hanxiao Wang, Yuan-Chen Guo, Ying-Tian Liu, Zi-Xin Zou, Biao Zhang, Weize Quan, Ding Liang, Yan-Pei Cao, and Dong-Ming Yan. Face: A face-based autoregressive representation for high-fidelity and efficient mesh generation. *arXiv preprint arXiv:2603.01515*, 2026a. 1, 2, 4, 6, 8, 20

Hanxiao Wang, Ying-Tian Liu, Yuan-Chen Guo, Qi-Yuan Feng, Zi-Xin Zou, Ding Liang, Biao Zhang, and Yan-Pei Cao. Nexus: Native mesh generation with diffusion. *ACM Transactions on Graphics (TOG)*, 45(4):1–14, 2026b. 1, 2

Yuxuan Wang, Xuanyu Yi, Haohan Weng, Qingshan Xu, Xiaokang Wei, Xianghui Yang, Chunchao Guo, Long Chen, and Hanwang Zhang. Nautilus: Locality-aware autoencoder for scalable mesh generation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 10961–10970, 2025b. 2

Haohan Weng, Yikai Wang, Tong Zhang, CL Chen, and Jun Zhu. Pivotmesh: Generic 3d mesh generation via pivot vertices guidance. *arXiv preprint arXiv:2405.16890*, 2024. 2

Haohan Weng, Zibo Zhao, Biwen Lei, Xianghui Yang, Jian Liu, Zeqiang Lai, Zhuo Chen, Yuhong Liu, Jie Jiang, Chunchao Guo, et al. Scaling mesh generation via compressive tokenization. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 11093–11103, 2025. 1, 2, 7

Jianfeng Xiang, Zelong Lv, Sicheng Xu, Yu Deng, Ruicheng Wang, Bowen Zhang, Dong Chen, Xin Tong, and Jiaolong Yang. Structured 3d latents for scalable and versatile 3d generation. *arXiv preprint arXiv:2412.01506*, 2024. 18, 20

Rui Xu, Tianyang Xue, Qiujie Dong, Le Wan, Zhe Zhu, Peng Li, Zhiyang Dou, Cheng Lin, Shiqing Xin, Yuan Liu, et al. Meshmosaic: Scaling artist mesh generation via local-to-global assembly. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 20003–20013, 2026. 2

Dingdong Yang, Jian Liu, Biwen Lei, Haohan Weng, Zhuo Chen, Song Guo, Hao Richard Zhang, Ali Mahdavi Amiri, and Chunchao Guo. Meshfim: Local low-poly mesh editing via fill-in-the-middle autoregressive generation. *arXiv preprint arXiv:2605.08744*, 2026. 3

Ruowen Zhao, Junliang Ye, Zhengyi Wang, Guangce Liu, Yiwen Chen, Yikai Wang, and Jun Zhu. Deepmesh: Auto-regressive artist-mesh creation with reinforcement learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 10612–10623, 2025. 2

Tianhao Zhao, Youjia Zhang, Hang Long, Jinshen Zhang, Wenbing Li, Yang Yang, Gongbo Zhang, Jozef Hladkỳ, Matthias Nießner, and Wei Yang. Lato: 3d mesh flow matching with structured topology preserving latents. *arXiv preprint arXiv:2603.06357*, 2026. 1, 2

Zibo Zhao, Wen Liu, Xin Chen, Xianfang Zeng, Rui Wang, Pei Cheng, BIN FU, Tao Chen, Gang YU, and Shenghua Gao. Michelangelo: Conditional 3d shape generation based on shape-image-text aligned latent representation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=xmxgMij3LY. 4

APPENDIX

APPENDIX OVERVIEW

The appendix provides additional details of our method, implementation, applications, and experimental results.

- **Appendix A** introduces the Trie data structure, its basic operations, and its use for frontier management in TrieMesh.
- **Appendix B** provides additional training and evaluation details, including the learning-rate schedule, progressive sampling, and point-cloud sampling.
- **Appendix C** presents the complete mesh serialization algorithm, its time and space complexity, and empirical statistics of the frontier and Trie sizes.
- **Appendix D** describes the alternative strategies for handling residual frontiers and analyzes the class imbalance and error propagation associated with learned routing.
- **Appendix E** presents image-conditioned mesh generation results and an extended generation gallery.
- **Appendix F** details the mesh hole-filling application, including its problem setting, implementation, baseline configurations, additional results, and interactive Gradio interface.
- **Appendix G** discusses the limitations of virtual-face termination and potential extensions of explicit frontier representations to diffusion-based mesh generation.

A DETAILS OF THE TRIE DATA STRUCTURE

This section introduces the trie data structure through a concrete example and details its construction and basic operations, including insertion, deletion, update, and search. The term "Trie" was introduced by Fredkin (Fredkin, 1960) and is derived from the word "re**trie**val", reflecting its use for efficient key retrieval. Given a key $k = (s_1, s_2, \ldots, s_d)$, each symbol corresponds to one level of the tree, and the path from the root to a node represents a prefix of the key. Keys with common prefixes therefore share the same path instead of storing the repeated symbols independently. For clarity, Fig. 8 illustrates the Trie using the string keys `cad`, `car`, and `cat`, while the complete operations are summarized in Algorithm 1.

Node Structure and Initialization. Each Trie node u contains a child map $u.\mathrm{child}$ and a terminal flag $u.\mathrm{terminal}$. The child map associates each possible symbol with the corresponding successor node. The terminal flag indicates whether a complete key ends at u. This flag is necessary because a node may represent both a complete key and the prefix of a longer key.

The NEWNODE operation initializes the child map as empty and sets the terminal flag to false. A Trie $\mathcal{T}$ is then initialized by creating a single empty root node. The root does not correspond to a symbol and only serves as the common starting point for all stored keys. In Fig. 8, red markers denote nodes whose terminal flags are true.

Tree Construction and Insertion. A Trie is constructed by inserting its keys one at a time. To insert a key k, the traversal begins at the root and processes the symbols of k sequentially. If the current node already contains a child associated with the next symbol, that child is reused. Otherwise, a new node is created and attached to the current node. After processing the final symbol, the terminal flag of the resulting node is set to true.

For example, the left panel of Fig. 8 shows a Trie initially constructed from `car` and `cad`. The two keys share the prefix `ca`, so the nodes corresponding to `c` and `a` are stored only once. Their final symbols, `r` and `d`, form two separate branches. When `cat` is subsequently inserted, as shown in the middle panel, the existing path for `ca` is reused and only a new child node for `t` is created.

Search. Searching for a key follows the same root-to-node traversal used during insertion. Starting from the root, the algorithm queries the child map using each symbol of the input key. If a required

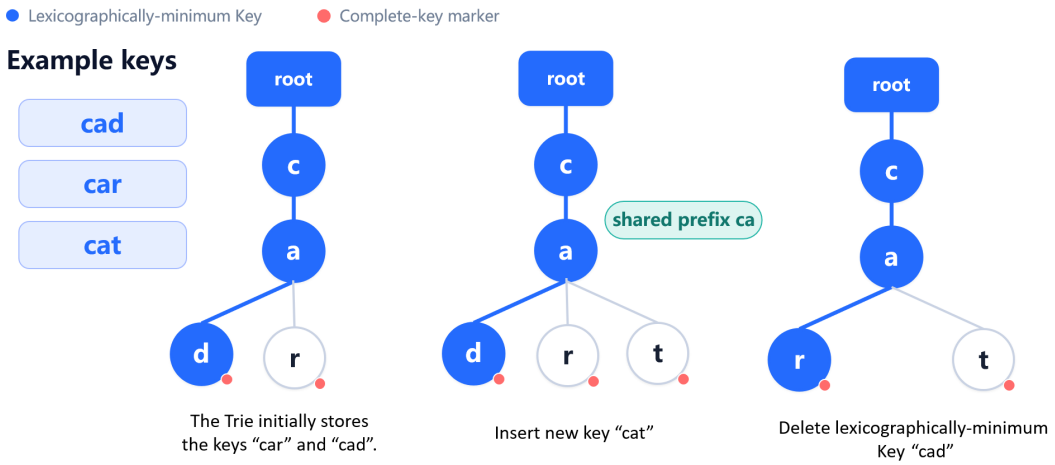

Figure 8: Illustration of Trie construction and key updates. The Trie is initialized with `car` and `cad`. Inserting `cat` reuses the shared prefix `ca`, whereas deleting `cad` removes its terminal marker and prunes the unused branch. Blue highlights the lexicographically smallest complete key, and red markers indicate terminal nodes.

child is missing, the key is not stored in the Trie and the search immediately returns false. If all symbols are found, the search returns the terminal flag of the final node.

Checking the terminal flag distinguishes exact-key search from prefix search. Reaching the node corresponding to `ca`, for example, confirms that `ca` is a valid prefix, but does not imply that `ca` is itself a stored key. An exact search succeeds only when the final node is marked as terminal.

When child symbols are maintained in lexicographic order, the Trie also supports deterministic retrieval of the lexicographically smallest key. A depth-first traversal visits the smallest available child first and returns when it reaches a terminal node. In Fig. 8, the blue path indicates the current lexicographically smallest complete key.

Deletion. The deletion procedure first searches for the target key. If the key does not exist, the Trie remains unchanged and the operation returns false. Otherwise, the algorithm follows its path and clears the terminal flag of the final node. This logical deletion is sufficient to ensure that subsequent exact searches no longer return the deleted key.

For simplicity, Algorithm 1 presents logical deletion without node pruning. A practical implementation may additionally remove unused nodes by traversing the deleted path backward. A node can be removed when its terminal flag is false and it has no remaining children. The backward traversal stops as soon as it encounters a terminal node or a node shared by another key. The right panel of Fig. 8 shows this compacted form. After deleting `cad`, the unused `d` branch is removed, while the shared prefix `ca` and the branches for `car` and `cat` are preserved. Consequently, `car` becomes the new lexicographically smallest key.

Update. A Trie does not require a separate structural operation for modifying a key. Updating k_{old} to k_{new} is implemented by deleting the old key and inserting the new key. If k_{old} is not present, the update returns false and leaves the Trie unchanged. Otherwise, the deletion and insertion procedures automatically preserve any prefixes shared with the remaining keys.

Complexity and Use in TrieMesh. Let d denote the length of a key. Assuming constant-time child access over a fixed symbol vocabulary, insertion, exact search, and logical deletion each require $O(d)$ time. Updating a key requires $O(d_{\text{old}} + d_{\text{new}})$ time. Retrieving the lexicographically smallest key also takes $O(d)$ time when the smallest child can be accessed directly. The space complexity is proportional to the number of distinct prefixes across all stored keys.

In TrieMesh, each frontier edge is represented by the canonical length-six key $(z_i, y_i, x_i, z_j, y_j, x_j)$. Therefore, every active frontier corresponds to a terminal root-to-node path of fixed depth $d = 6$. Inserting a key records a newly exposed frontier, while deleting a key removes a closed frontier.

Algorithm 1 Trie Data Structure and Basic Operations

NewNode()
Allocate a new Trie node u
u.child $\leftarrow \emptyset$ {Empty child map}
u.terminal $\leftarrow$ false {No key ends at u}
return u

InitializeTrie()
$\mathcal{T}$.root $\leftarrow$ NEWNODE()
return $\mathcal{T}$

Insert($\mathcal{T}, k$)
$u \leftarrow \mathcal{T}$.root
for each symbol s in k **do**
  if $s \notin u$.child **then**
    u.child[s] $\leftarrow$ NEWNODE()
  end if
  $u \leftarrow u$.child[s]
end for
u.terminal $\leftarrow$ true
return $\mathcal{T}$

Search($\mathcal{T}, k$)
$u \leftarrow \mathcal{T}$.root
for each symbol s in k **do**
  if $s \notin u$.child **then**
    return false
  end if
  $u \leftarrow u$.child[s]
end for
return u.terminal

Delete($\mathcal{T}, k$)
if SEARCH($\mathcal{T}, k$) = false **then**
  return false
end if
$u \leftarrow \mathcal{T}$.root
for each symbol s in k **do**
  $u \leftarrow u$.child[s]
end for
u.terminal $\leftarrow$ false {Logical deletion}
return true

Update ($\mathcal{T}, k_{\text{old}}, k_{\text{new}}$)
if DELETE ($\mathcal{T}, k_{\text{old}}$) **then**
  INSERT ($\mathcal{T}, k_{\text{new}}$)
  return true
end if
return false

Maintaining children in lexicographic order allows TrieMesh to retrieve the next expansion frontier deterministically without scanning all active frontiers.

B More Implementation Details

Learning Rate Schedule. We first train on the full dataset using a learning-rate schedule with a 5K-step linear warm-up to 10^{-4}, followed by a constant learning rate until 50K steps and cosine annealing over the final 50K steps.

Progressive Sampling. To alleviate the long-tailed distribution of mesh complexity (see Fig. 9), we adopt the progressively balanced sampling strategy introduced in MeshSilkSong (Song et al., 2026). Specifically, we partition the training meshes into face-count bins with a bin width of 500

Algorithm 2 Mesh Serialization via Trie Traversal

Require: Mesh vertices V, real faces F
Ensure: Ordered face sequence S

$(V, F) \leftarrow \textsc{SortVerticesAndFaces}(V, F)$
Build raw edge-to-face incidence $\mathcal{A}_{\text{raw}}$
$(\mathcal{A}_{\text{pair}}, \mathcal{I}) \leftarrow \textsc{BuildPairAwareAdjacency}(\mathcal{A}_{\text{raw}})$
$\mathcal{F}_{\text{vis}} \leftarrow \emptyset$
$S \leftarrow [\,]$
Initialize an empty Trie $\mathcal{T}$
while $|\mathcal{F}_{\text{vis}}| < |F|$ **do**
  $f \leftarrow$ lexicographically smallest unvisited face
  Append f to S and add it to $\mathcal{F}_{\text{vis}}$
  for all $\hat{e} \in \mathcal{I}(f)$ **do**
    Insert $\hat{e}$ into $\mathcal{T}$
  end for
  while $|\mathcal{T}| > 0$ **do**
    $\hat{e} \leftarrow \min_{\prec}(\mathcal{T})$ {Edge key first; pair index breaks ties}
    $e = (v_0, v_1) \leftarrow \textsc{UnderlyingEdge}(\hat{e})$
    $\mathcal{C} \leftarrow \mathcal{A}_{\text{pair}}(\hat{e}) \setminus \mathcal{F}_{\text{vis}}$ {At most one unvisited paired face}
    if $\mathcal{C} \neq \emptyset$ **then**
      $f' \leftarrow$ the unique face in $\mathcal{C}$
      Reorder $f' = (v_0, v_1, v_2)$ according to e
      Append f' to S and add it to $\mathcal{F}_{\text{vis}}$
      for all $\hat{e}' \in \mathcal{I}(f')$ **do**
        $\textsc{ToggleInstance}(\mathcal{T}, \hat{e}')$ {Delete if active; otherwise insert}
      end for
    else
      Append (v_0, v_1, EOS) to S
      Delete $\hat{e}$ from $\mathcal{T}$ {Consume an unmatched instance}
    end if
  end while
end while
return S

faces and progressively adjust the sampling distribution during training. The sampling probability gradually shifts from the original data distribution toward a more balanced distribution across different face-count bins, allowing the model to better cover meshes of varying complexity.

Point-cloud Sampling. For each augmented mesh, we sample 50K surface points with normals and add Gaussian noise with $\sigma = 0.01$ with a probability of 0.5. We then subsample 8,192 points as the geometric condition. The point-cloud encoder is initialized from pretrained Michelangelo weights and jointly optimized with the autoregressive mesh decoder.

C Mesh Serialization Pseudocode and Complexity Analysis

Pseudocode. The complete serialization procedure is given in Algorithm 2. We first sort the vertices and faces lexicographically (Siddiqui et al., 2024; Chen et al., 2024b) and construct the pair-aware adjacency described above. Each face-edge incidence belongs to one pair-aware instance. Instances sharing the same geometric edge are stored under the same Trie key and ordered by their pair indices.

The smallest unvisited face initializes each traversal component. At each step, we select the smallest active instance and visit its unique unvisited paired face, if one exists. The new face is appended to the sequence, after which its active edge instances are removed and its inactive instances are inserted. If no real partner remains, a virtual face (v_0, v_1, EOS) consumes the selected instance. When the Trie becomes empty, traversal restarts from the smallest remaining unvisited face until all real faces have been serialized.

Complexity of Trie-based Face Traversal. Let $M = |F|$ denote the number of real faces, and let $M_{\text{vir}} = B + N_{\text{odd}}$ denote the number of virtual faces introduced during serialization. Although Algorithm 2 contains nested loops, its total work is linear in the number of processed faces and residual frontiers: each real face is processed exactly once, and each residual frontier generates at most one virtual face. Because every real face is triangular, processing a face requires updating at most three frontier edges and therefore takes $O(1)$ time. In addition, the Trie has a fixed depth of six, corresponding to the quantized coordinates of the two frontier vertices. Consequently, insertion, deletion, and lexicographically minimum retrieval all require $O(1)$ time. The overall serialization time complexity is therefore

$$O(M + M_{\text{vir}}).$$

To characterize peak auxiliary memory, let A denote the maximum number of simultaneously active frontier edges, and let T denote the maximum number of Trie nodes during traversal. Since each active frontier is represented by a fixed-depth key, it can introduce only a constant number of Trie nodes. Hence, $T = O(A)$, and the Trie requires $O(A)$ auxiliary space.

The empirical distributions in Figs. 11 and 12 characterize the peak working-set sizes of these structures. As shown in Fig. 11, A/M is strongly right-skewed: most training examples exhibit low normalized frontier counts, while the sparse tail extends to approximately 0.5. Similarly, Fig. 12 shows that T/M is concentrated at small values but has a broader tail, with rare examples reaching approximately 2. This wider range is consistent with the fact that a single active frontier may occupy multiple Trie nodes. Overall, these results indicate favorable practical memory constants for most examples, although the peak Trie size can occasionally exceed M by a constant factor.

Including the serialized output sequence, the total space complexity is

$$O(M + M_{\text{vir}} + T) = O(M + M_{\text{vir}} + A).$$

Under the usual linear bound $A = O(M)$, the auxiliary Trie space is linear in the number of real faces. Furthermore, as shown in Fig. 10, the virtual-face ratio M_{vir}/M is strongly concentrated near zero in the training set, indicating that virtual faces incur only a small relative overhead in practice. Thus, for the evaluated data, M_{vir} scales proportionally with M, and both the serialization time and total space usage exhibit effective linear scaling with respect to the number of real faces, i.e., $O(M)$ in practice.

Trie versus Heap. A min-heap can efficiently retrieve the lexicographically smallest frontier, but frontier management requires more than minimum selection. After generating a face, each candidate edge must be queried and either inserted or deleted according to whether it is already active. For n active frontiers, a heap supports insertion and minimum extraction in $O(\log n)$ time, but searching for or deleting an arbitrary key requires $O(n)$ time unless an auxiliary index is maintained. A heap combined with a hash set and lazy deletion can improve these operations, but duplicates the frontier state and may retain stale entries.

In contrast, a Trie supports key lookup, insertion, deletion, and minimum-key retrieval in $O(d)$ time. Since each frontier key in our representation has a fixed length of $d = 6$, these operations are independent of the number of active frontiers. Shared prefixes also reduce repeated storage of common coordinate prefixes, although the actual memory cost depends on the implementation.

D Residual Frontier Handling Strategies

To assess the virtual-face design, we compare it with two learned-routing alternatives. Rather than inserting virtual faces to resolve residual frontiers, these alternatives predict an explicit frontier-routing code after each generated face.

Routing target. For each generated triangular face f_t, let

$$\mathbf{r}_t = (r_{t,1}, r_{t,2}, r_{t,3}) \in \{0,1\}^3$$

denote its 3-bit routing mask, where correspond to the three edges of f_t in a fixed canonical order. We encode this mask as a discrete routing tokens

$$c_t = 4r_{t,1} + 2r_{t,2} + r_{t,3} \in \mathcal{V}_{\text{route}} = \{0, 1, \ldots, 7\}.$$

Thus, routing tokens 0 through 7 correspond to the binary masks 000 through 111, respectively. The routing vocabulary is independent of the face-token vocabulary.

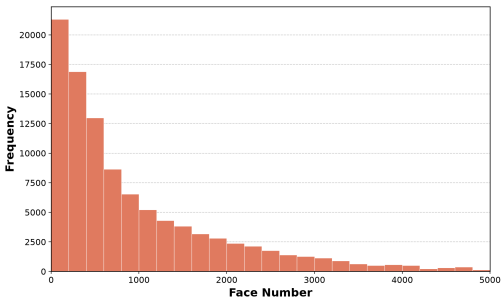

Figure 9: Face-count distribution of the training meshes, showing a long-tailed distribution over mesh complexity.

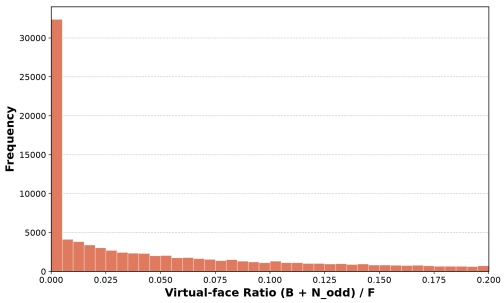

Figure 10: Distribution of virtual-face overhead in the training set, measured as $(B + N_{\text{odd}})/F$. Most meshes exhibit low overhead, with a long tail toward higher ratios.

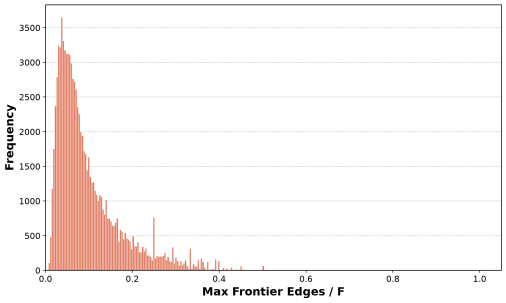

Figure 11: Frequency distribution of the maximum number of frontier edges normalized by F.

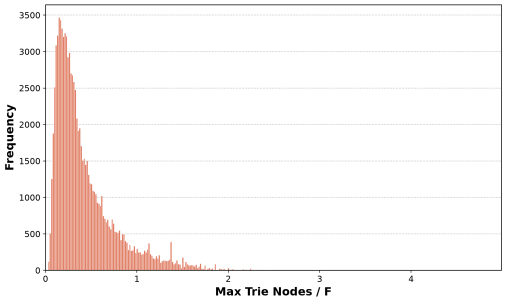

Figure 12: Frequency distribution of the maximum number of trie nodes normalized by F.

Parallel routing. The nine-layer face cascade and an independent routing head operate as parallel branches: the former predicts the nine face tokens, while the latter predicts the routing tokens in $\mathcal{V}_{\text{route}}$. The predicted routing token directly determines the frontier update.

Cascaded routing. We extend the original nine-layer cascade with a tenth layer dedicated to routing prediction. This additional layer predicts a routing code from the vocabulary $\mathcal{V}_{\text{route}}$. Thus, the routing is incorporated into the intra-face cascade.

Analysis. Although the main paper reports that learned routing performs poorly, we further analyze the underlying cause. As shown in Fig. 13, the routing labels are extremely imbalanced: 94.18% of faces have state 000, indicating that none of their edges is adjacent to a virtual face, whereas state 111, in which all three edges are adjacent to virtual faces, accounts for only 0.09% of faces on average. Consequently, the routing objective is dominated by the majority class, allowing a model to achieve high token-level accuracy by favoring the 000 state while receiving limited supervision for the rare but structurally important routing patterns.

Moreover, routing errors have a disproportionate effect on autoregressive decoding. An incorrect routing token causes an incorrect frontier update, such as inserting an edge that should be excluded from the Trie or omitting an edge that should remain active. The resulting frontier state is then used to condition subsequent face generation, causing the error to propagate and potentially compound over later decoding steps. This combination of highly imbalanced labels and error accumulation helps explain the inferior performance of learned routing compared with virtual faces, which encode frontier termination explicitly and update the frontier deterministically.

E Image-Conditioned Mesh Generation and Gallery

For image-conditioned generation, we use TRELLIS (Xiang et al., 2024) to reconstruct an intermediate 3D shape and uniformly sample 8,192 surface points as the geometric condition for TrieMesh.

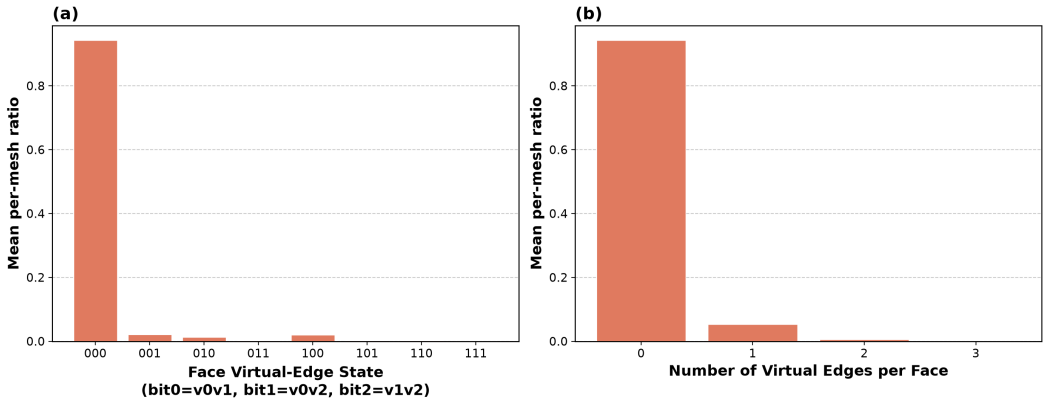

Figure 13: Mean per-mesh ratios of virtual-face adjacency states. (a) Each 3-bit state indicates whether the face edges (v_0v_1, v_0v_2, v_1v_2) are adjacent to virtual faces. (b) Number of such edges per face. The dominance of state 000 reveals the severe class imbalance of the learned routing labels.

The final mesh connectivity and triangulation are generated autoregressively by our model. Representative results are shown in Fig. 14, with additional examples provided in the gallery in Fig. 15.

F Implementation Details for Mesh Hole Filling

F.1 Problem Setting

Let $\mathcal{M} = (V, F)$ be a triangular mesh. During training, we remove a connected face patch $F_{\text{miss}} \subset F$ and use the remaining faces

$$F_{\text{remain}} = F \setminus F_{\text{miss}} \tag{10}$$

as the observed incomplete mesh.

For an edge e and a face subset $F' \subseteq F$, let $\mathcal{I}_{F'}(e)$ denote the faces in F' incident to e. We define the boundary between the observed and missing regions as

$$\mathcal{B} = \{e \in E \mid \mathcal{I}_{F_{\text{remain}}}(e) \neq \emptyset \ \wedge \ \mathcal{I}_{F_{\text{miss}}}(e) \neq \emptyset\}. \tag{11}$$

This definition only requires an edge to be incident to both regions and therefore does not assume that the input mesh is manifold.

The objective is to generate a face patch $\widehat{F}_{\text{miss}}$ that fills the missing region and forms the completed mesh

$$\widehat{\mathcal{M}} = \left(V \cup \widehat{V}, F_{\text{remain}} \cup \widehat{F}_{\text{miss}}\right), \tag{12}$$

where $\widehat{V}$ contains any newly generated vertices.

For training, the remaining faces form a visible prefix, while the missing faces form the prediction target:

$$S = \left[F^{\text{ZYX}}_{\text{remain}} \parallel F^{\text{Trie}}_{\text{miss}}\right]. \tag{13}$$

F.2 Implementation Details

Training Hole Construction. We construct a connected missing patch on the largest face-connected component of each training mesh. Starting from a uniformly sampled seed face, the patch is expanded by breadth-first search until it contains between 5% and 10% of the component. We retain only patches for which each hole-boundary edge is incident to exactly one remaining face and one missing face, each internal patch edge is incident to two missing faces, and the patch does not intersect a pre-existing open boundary. The patch may contain multiple boundary loops. When data augmentation is enabled, both the mesh transformation and the missing patch are resampled at every training iteration.



Figure 14: Image-conditioned mesh generation. Given an input image, TRELLIS (Xiang et al., 2024) first reconstructs a dense 3D mesh; surface points sampled from this mesh then provide the geometric condition for our generator.

Completion Sequence Construction. The remaining faces are globally sorted in ZYX order (Siddiqui et al., 2024; Wang et al., 2026a) and placed at the beginning of the sequence as a bidirectionally visible prefix. The missing faces form the autoregressive completion target.

To construct the target sequence, the hole-boundary edges initialize the frontier Trie. At each step, we select the lexicographically smallest frontier edge, append its unique unvisited incident face from the missing patch, and update the Trie using the same XOR rule as in TrieMesh. The process continues until all missing faces have been visited and the Trie becomes empty. Because the initial boundary is already provided, no seed face or virtual face is included in the completion target.

Trie-guided Inference. At inference time, the remaining faces are encoded as a bidirectionally visible prefix, and the specified target-boundary edges initialize the frontier Trie.



Figure 15: Gallery of mesh generation results produced by TrieMesh.

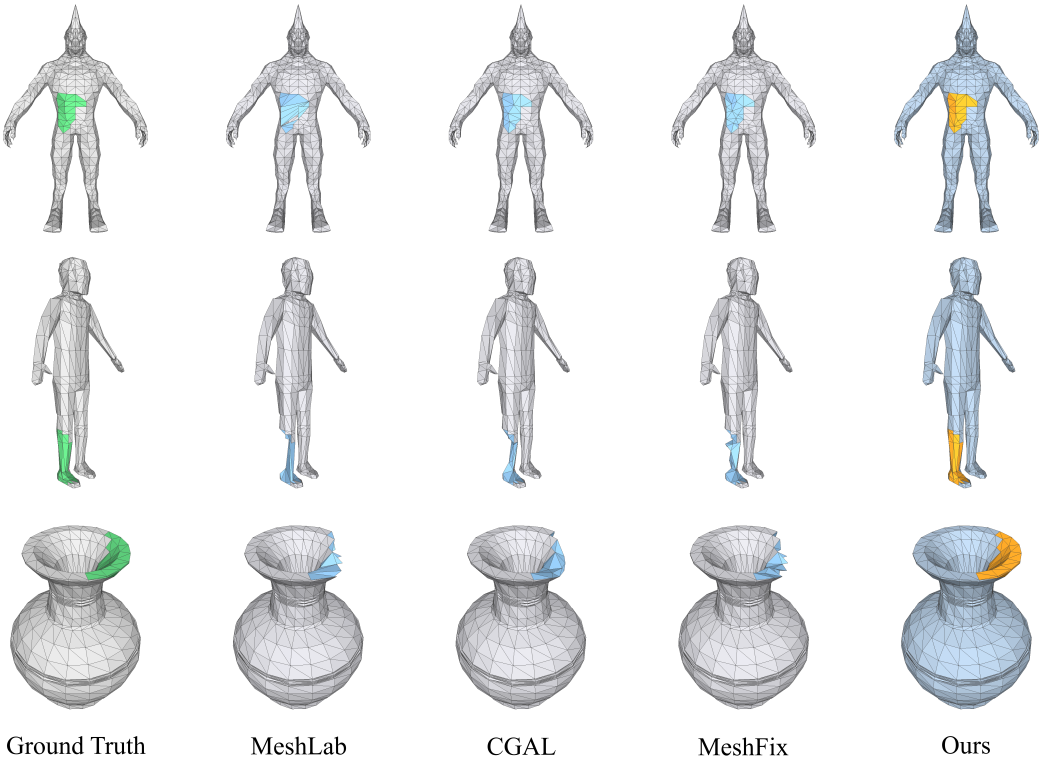

Figure 16: Additional qualitative comparisons of mesh hole filling. Colored regions mark the hole-filling area. Our method produces completions that better follow the structure and triangulation patterns of the remaining mesh.

At each step, the lexicographically smallest frontier edge (v_0, v_1) is selected. Its six coordinate tokens are provided directly to the decoder, which predicts only the completion vertex v_2. After the triangle (v_0, v_1, v_2) is generated, each of its edges is used to update the Trie: an edge already in the Trie is removed, while a new edge is inserted. Generation terminates when the Trie becomes empty, without starting a new seed face or introducing virtual faces.

Table 6: Quantitative comparison of mesh hole filling on 44 test cases. Our method achieves the lowest Chamfer and Hausdorff distances and the highest orientation-invariant normal consistency.

Method	Chamfer ↓	Hausdorff ↓	$\lvert NC\rvert$ ↑
PyMeshLab	0.032	0.085	0.791
CGAL	0.018	0.061	0.899
PyMeshFix	0.020	0.046	0.880
Ours	**0.009**	**0.026**	**0.940**

F.3 Baseline Settings

We compare against three famous hole-filling implementations: PyMeshLab (Cignoni et al., 2008), PyMeshFix (Attene, 2010), and CGAL (The CGAL Project, 2026). All methods receive the same incomplete triangular mesh and do not access the removed ground-truth patch.

PyMeshLab. We apply MeshLab's *Repair Non Manifold Edges* filter before *Close Holes*. The latter uses a maximum hole size of 100, with self-intersection avoidance and hole refinement enabled.

PyMeshFix. We call `PyTMesh.fill_small_boundaries` with `nbe=0` so that all detected boundary loops are processed, and enable `refine=True` to refine the triangulation of the inserted patches. We do not invoke the global cleaning or repair routines, ensuring that the comparison evaluates hole filling rather than complete mesh sanitization. Setting `nbe=0` instructs PyMeshFix to process all detected boundary loops. We do not invoke its global `clean` or `repair` stages, ensuring that the comparison evaluates hole filling rather than full mesh sanitization.

CGAL. We first process the input using `orient_polygon_soup` and then fill each detected border cycle with `triangulate_refine_and_fair_hole`. This routine applies CGAL's triangulation, refinement, and fairing pipeline. We set `density_control_factor` to $\sqrt{2}$ and `fairing_continuity` to 1, corresponding to C^1 fairing.

Evaluation. Only the reconstructed patch is evaluated against the corresponding ground-truth patch. Faces from the observed input mesh are excluded from the patch-level metrics.

F.4 Additional Hole Filling Results

The quantitative results in Tab. 6 show that our method achieves the lowest Chamfer and Hausdorff distances and the highest orientation-invariant normal consistency over the 44 test cases. Additional qualitative comparisons are presented in Fig. 16. Unlike classical methods that primarily construct patches from local boundary geometry, our generative approach can use the remaining mesh as context and recover triangulation patterns commonly found in artist-created meshes.

In the first row, our completion follows the triangulation density and connectivity of the observed surface and forms a more balanced pattern with the intact right side of the abdomen. This example illustrates that the model can adapt its completion to the existing tessellation instead of merely producing a geometrically valid local patch. The second row contains a large missing region covering most of the left lower leg. The classical methods produce incomplete or collapsed patches because the hole provides insufficient local geometric constraints. Our method instead uses the intact right leg as a strong contextual cue and reconstructs a coherent missing part. In the third row, the observed rim exhibits a clear triangle-strip pattern. Our completion continues this pattern across the missing region, whereas the classical methods produce less regular local triangulations. Triangle-strip layouts are a common style in artist-created meshes and are therefore widely represented in our training data. This enables the model to learn their characteristic connectivity and reproduce it when completing mesh regions with compatible triangulation patterns.

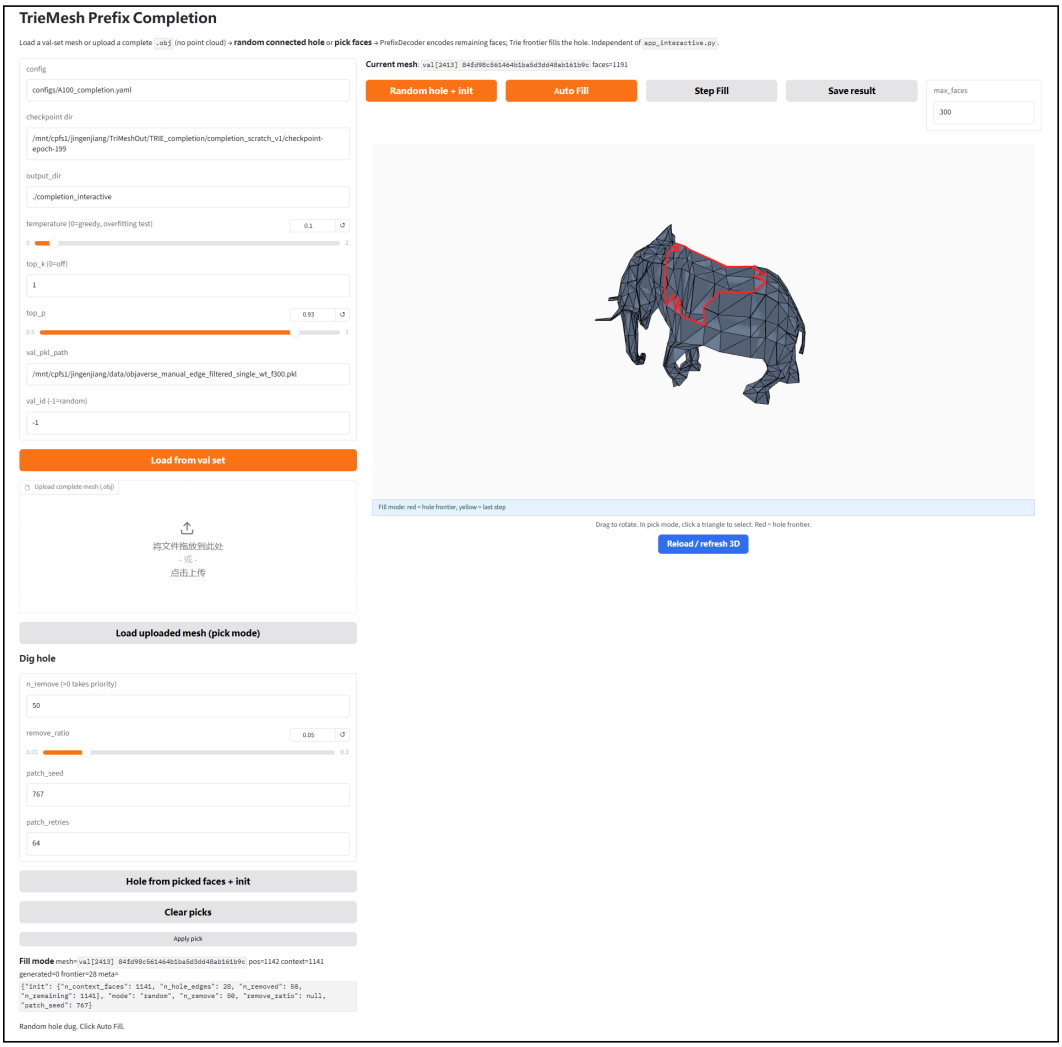

Figure 17: Interactive Gradio interface for mesh hole filling. The left panel provides mesh loading, hole construction, and sampling controls, while the right panel supports stepwise or automatic completion and visualizes the evolving mesh and its active Trie frontiers.

F.5 Interactive Gradio Interface

We provide an interactive Gradio interface for demonstrating the mesh hole-filling application, as shown in Fig. 17. The interface operates directly on the remaining mesh and does not require point-cloud conditioning. Its left panel contains controls for loading meshes, constructing holes, and configuring the sampling strategy, while the right panel provides the completion controls and an interactive Three.js visualization.

Users can load a mesh from the validation set or upload a complete OBJ file. A missing region can then be created either by randomly growing a connected face patch or by selecting triangles directly in the 3D viewer. After the selected faces are removed, the remaining faces form the bidirectional prefix and the boundary edges of the missing region initialize the Trie. The interface supports both stepwise and automatic completion. `Step Fill` expands the lexicographically smallest frontier by one triangle, whereas `Auto Fill` repeats this process until the Trie becomes empty or the specified face budget is reached. Standard sampling parameters, including temperature, top-k, and top-p, can be adjusted interactively.

The 3D viewer supports rotation and zooming and visualizes the mesh state throughout completion. Active frontier edges are highlighted in red, and the most recently generated face is shown in yellow,

making the progression of Trie-guided completion easy to inspect. The interface also reports the number of generated faces, the number of remaining frontier edges, and the termination status. Completed results can be exported together with the remaining mesh, generated patch, removed ground-truth patch, colored visualization, and the corresponding completion metadata.

G Limitations and Future Work

TrieMesh uses virtual-face tokens to terminate residual frontiers caused by open boundaries and odd-degree non-manifold edges. Although their overhead is moderate on the evaluated datasets, meshes containing many such structures may require longer sequences and additional decoding steps. Future work could explore token-free frontier termination while preserving deterministic and reliable Trie updates.

TrieMesh currently selects frontier edges using a fixed lexicographic rule. Because the active frontier is explicit and queryable, it could also serve as a control interface for interactive systems or external editing agents: the frontier set provides the current structural state, and selecting an edge determines where the mesh expands next. An important direction is to support such controllable expansion while maintaining valid frontier updates and reliable termination.