GetMesh: A Controllable Model for High-quality Mesh Generation and Manipulation

Zhaoyang Lyu[1⋆] Ben Fei[1,2⋆] Jinyi Wang[3⋆] Xudong Xu[1]
Ya Zhang[1,3] Weidong Yang[2] Bo Dai[1]

[1] Shanghai Artificial Intelligence Laboratory
[2] Fudan University
[3] Shanghai Jiao Tong University

Abstract. Mesh is a fundamental representation of 3D assets in various industrial applications, and is widely supported by professional softwares. However, due to its irregular structure, mesh creation and manipulation is often time-consuming and labor-intensive. In this paper, we propose a highly controllable generative model, `GetMesh`, for mesh generation and manipulation across different categories. By taking a varying number of points as the latent representation, and re-organizing them as triplane representation, `GetMesh` generates meshes with rich and sharp details, outperforming both single-category and multi-category counterparts. Moreover, it also enables fine-grained control over the generation process that previous mesh generative models cannot achieve, where changing global/local mesh topologies, adding/removing mesh parts, and combining mesh parts across categories can be intuitively, efficiently, and robustly accomplished by adjusting the number, positions or features of latent points. Project page is https://getmesh.github.io.

Keywords: 3D Generation · Controllable Generation · Diffusion Model

1 Introduction

3D asset generation is of significant value for AR/VR, gaming, filming and design. Meshes are a kind of fundamental representation for 3D assets in industrial applications since their creation, editing, and rendering are supported by many professional softwares. However, mesh creation and editing can be quite time-consuming and require extensive artistic training, since meshes are irregular in their data structure with varying topologies across different instances and categories. Therefore, a controllable paradigm that supports intuitive and efficient generation and manipulation of meshes is of great need.

While directly operating on meshes is difficult and sometimes infeasible, an alternative is to rely on a suitable latent representation and associate it with mesh via a pair of encoder and decoder. Researchers have designed many types of latent representations for mesh generation. Among them, point-based representations [21, 43] are compact and convenient for editing, but rely on a Poisson reconstruction-like algorithm [29] to obtain meshes, which are overly smooth

⋆ Equal Contribution.

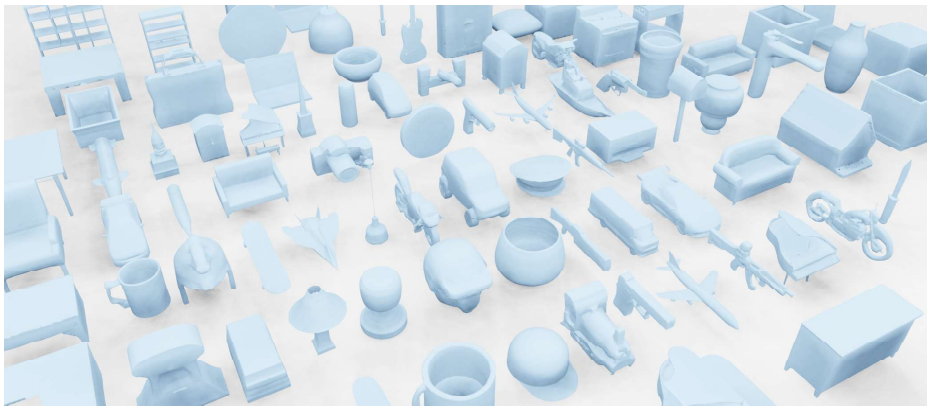

Fig. 1: Meshes generated by our method. GetMesh is able to generate diverse and high-quality meshes across the 55 categories in ShapeNet.

without sharp details. Voxel-based representations [15,17,46,47] are proposed for 3D shapes due to their regular data structure, yet their computational and memory cost increase significantly for high-quality meshes. Finally, triplane-based representations [8, 9, 35] are compact and efficient for high-quality mesh modeling, but can be quite obscure and inefficient to control as they are squeezed along a specific dimension. Therefore, despite the blooming attention and progress in text-to-3D [3, 16, 30, 39, 40] generation, finding a suitable latent representation and subsequently an intuitive and efficient paradigm for controllable mesh generation and manipulation remains an open question.

In this paper, we propose a novel model, `GetMesh`, a multi-category generative model that enables both high-quality mesh generation and flexible control over the generation process. It combines the merits of both point-based representation and triplane-based representation, which is achieved by using *a varying number of points* as the latent representation, and *re-organizing them as triplane* representation. The feature and position distributions of these points can be respectively modeled by two diffusion models. A varying number of points as the latent representation provides significant controllability over the generation process, where changing global/local topologies of meshes, adding/removing mesh parts, as well as combining mesh parts across instances/categories can all be achieved intuitively, efficiently, and robustly by adjusting the number, positions, or features of these points. To avoid obtaining overly smoothed meshes from point-based representation, the proposed paradigm re-organizes these points by projecting their features onto *xy,xz,yz* planes according to their positions, forming a triplane-based representation. Subsequently, a triplane-based decoder with a refinement module can thus be used to extract high-quality meshes with sharp details.

We conduct extensive experiments on ShapeNet [2] to evaluate `GetMesh`. As expected, `GetMesh` generates meshes with rich and sharp details as shown in Figure 1, outperforming its multi-category counterparts, and even single-category generative models. Moreover, since `GetMesh` is capable of controlling mesh gen-

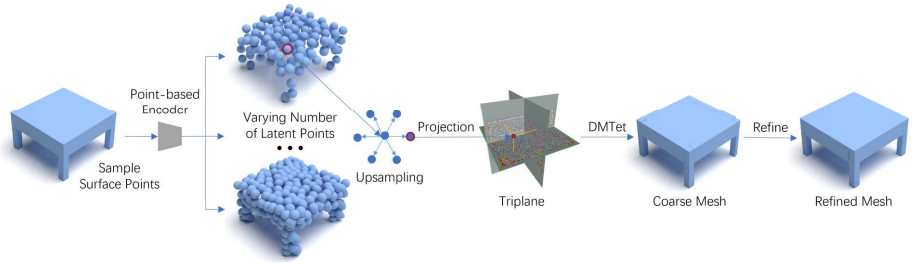

Fig. 2: Overview of the mesh autoencoder. Points are sampled from the surface of the input mesh and encoded to a varying number of latent points. The latent point representation is re-organized to the triplane representation by projecting the points to the triplane. DMTet is utilized to extract a coarse mesh from the triplane and a refinement module further refines the coarse mesh.

eration intuitively and flexibly, `GetMesh` is able to change the topology of generated meshes such as turning a twin-engine airplane into a four-engine one, and gradually turning a car into an airplane as shown in Section 5.6. `GetMesh` also successfully combines mesh parts across different categories, leading to a car with airplane wings, and a table with lamp top. Finally, `GetMesh` is shown to work well with an off-the-shelf material generative method to acquire materials for its generated meshes.

2 Background

Denoising Diffusion Probabilistic Models (DDPMs) are generative models that learn the distribution of samples in a dataset. A DDPM is composed of two processes: the diffusion process and the reverse process. The diffusion process gradually adds noise to clean samples $\boldsymbol{x}^0$ and turns them into Gaussian noises $\boldsymbol{x}^T$ after T steps. It is defined as

$$q(\boldsymbol{x}^1, \cdots, \boldsymbol{x}^T | \boldsymbol{x}^0) = \prod_{t=1}^{T} q(\boldsymbol{x}^t | \boldsymbol{x}^{t-1}), \tag{1}$$

$$\text{where } q(\boldsymbol{x}^t | \boldsymbol{x}^{t-1}) = \mathcal{N}(\boldsymbol{x}^t; \sqrt{1-\beta_t}\boldsymbol{x}^{t-1}, \beta_t \boldsymbol{I}), \tag{2}$$

$\mathcal{N}$ is the Gaussian distribution. In our experiments, we set $T = 1000$, and β_t linearly increase from 1×10^{-4} to 2×10^{-2} as t increases from 1 to T. The reverse process is the data generation process. It starts from a Gaussian noise $\boldsymbol{x}^T$ and denoises it step by step, eventually turning it into a clean sample $\boldsymbol{x}^0$. The reverse process is formally defined as

$$\begin{aligned} p_{\boldsymbol{\theta}}(\boldsymbol{x}^0, \cdots, \boldsymbol{x}^{T-1} | \boldsymbol{x}^T) &= \prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\boldsymbol{x}^{t-1} | \boldsymbol{x}^t), \\ \text{where } p_{\boldsymbol{\theta}}(\boldsymbol{x}^{t-1} | \boldsymbol{x}^t) &= \mathcal{N}(\boldsymbol{x}^{t-1}; \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}^t, t), \tilde{\beta}_t \boldsymbol{I}), \end{aligned} \tag{3}$$

$\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$. We follow [10] to reparameterize the mean $\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}^t, t)$ as

$$\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}^t, t) = \frac{1}{\sqrt{\alpha_t}} \left(\boldsymbol{x}^t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}^t, t) \right), \tag{4}$$

where $\alpha_t = 1 - \beta_t$, $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$, and $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}$ is a neural network with parameters specified by $\boldsymbol{\theta}$. The loss to train the network is

$$L(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x}^0 \sim p_{\text{data}}} \, \|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\sqrt{\bar{\alpha}_t}\boldsymbol{x}^0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\|^2, \tag{5}$$

where p_{data} is the distribution of the dataset, t is sampled uniformly between 1 and T, and $\boldsymbol{\epsilon}$ is a Gaussian noise.

3 Methodology

A mesh is represented as vertices and faces, where faces describe the connections between vertices. It is difficult to directly train generative models on meshes due to their irregular data structure and discrete connections between vertices. Therefore, we first train an autoencoder to encode a mesh to a point-based representation that is easier to process for neural networks. Then we train DDPMs in the latent space of the autoencoder with the same merit as [33,37], and high-quality meshes can be reconstructed from the generated latent representation by a decoder.

We first describe the architecture of the mesh autoencoder. Its overall architecture is shown in Figure 2. We sample points from the input mesh and use a point-based encoder to encode the points to a varying number of latent points and their features. Then we re-organize the latent point representation to a triplane representation. Finally, we use a triplane-based decoder with a refinement module to decode the triplane representation to a high-quality mesh. The detailed architecture of the mesh autoencoder is explained in the following sections.

3.1 Point-based Encoder

We sample a point cloud $\boldsymbol{a} \in \mathbb{R}^{N_{\text{in}} \times 3}$ from the surface of the input mesh and the encoder encodes the sampled point cloud to a set of latent points $\boldsymbol{x} \in \mathbb{R}^{N \times 3}$ with features $\boldsymbol{y} \in \mathbb{R}^{N \times D}$, where N_{in} is the number of input points, N is the number of latent points and D is the feature dimension. To enable more versatile editing like addition and deletion of latent points, we design an encoder that supports a varying number of latent points $\boldsymbol{x}$. The latent points are sampled using Furthest Point Sampling (FPS) from the input point cloud $\boldsymbol{a}$, and the number of latent points N is uniformly sampled in the interval $[N_{\text{min}}, N_{\text{max}}]$ during training, where $N_{\text{min}}, N_{\text{max}}$ are two hyperparameters that control the minimum and maximum number of latent points. The architecture of the encoder resembles the one proposed in [21], which is an improved version of PointNet++ [32] proposed in [20]. The encoder gradually downsamples the input point cloud and propagates features level by level, until features are propagated to the sampled latent points. We refer readers to the original paper [21] for details of the encoder and Appendix Section A.1 on how we encode the input point cloud to a varying number of latent points.

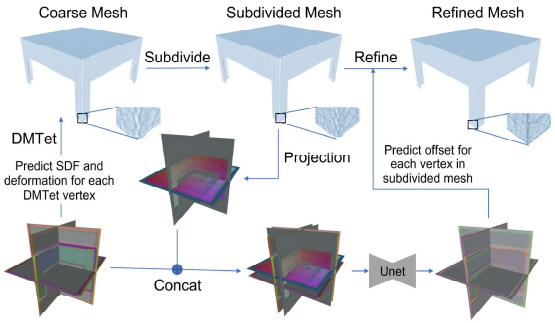

Fig. 3: Architecture of the refinement module.

3.2 Re-organize Latent Points as Triplane

In the previous section, we use a point-based encoder to encode the input mesh to a set of latent points and features. While this latent point representation is compact and intuitive to edit, it is difficult to reconstruct high-quality meshes with sharp details and thin structures directly from point-based representation using Poisson reconstruction or other related algorithms. In light of the recent success of triplane representation [8,9,35] in reconstructing relative high-quality meshes, we propose to re-organize the latent point representation as a triplane representation. The basic idea is to project the latent points together with their features onto the triplane. Specifically, for each point, we project it to the three perpendicular planes according to its position. Its feature is fed to a shared MLP and the output feature is assigned to the corresponding three pixels. Features of latent points projected to the same pixel are aggregated by average pooling. Pixels that do not correspond to any latent points are filled with zeros. In the point-based encoder, we set the number of latent points small in order to make the latent space compact. To make the triplane representation concrete, we first upsample the latent points $\boldsymbol{x}$ to a denser point cloud $\boldsymbol{x}_{\mathrm{u}}$, and propagate features from $\boldsymbol{x}$ to $\boldsymbol{x}_{\mathrm{u}}$. Then we project the upsampled point cloud $\boldsymbol{x}_{\mathrm{u}}$ to the triplane. More details of the upsampling process are provided in Appendix Section A.1.

3.3 Triplane-based Decoder

After re-organizing the latent point representation as the triplane representation, we use a triplane-based decoder together with a refinement module to reconstruct a high-quality mesh from the triplane representation. The architecture of the decoder is similar to [9]. The triplane is first processed by a UNet with 3D aware convolutions proposed in [38] and we obtain the processed triplane feature $\boldsymbol{f} \in \mathbb{R}^{3\times H\times W\times C}$. Next, DMTet [34] is utilized to extract a mesh differentiably. The SDF value and deformation of each vertex in the DMTet grid are predicted by querying the triplane feature $\boldsymbol{f}$. Specifically, for each vertex $\boldsymbol{v}$ in the DMTet grid, it is projected onto the triplane and we obtain three features $\boldsymbol{f}_{\boldsymbol{v}}^{\mathrm{xy}}, \boldsymbol{f}_{\boldsymbol{v}}^{\mathrm{yz}}, \boldsymbol{f}_{\boldsymbol{v}}^{\mathrm{zx}}$ through bilinear interpolation of the feature planes. The three features $\boldsymbol{f}_{\boldsymbol{v}}^{\mathrm{xy}}, \boldsymbol{f}_{\boldsymbol{v}}^{\mathrm{yz}}, \boldsymbol{f}_{\boldsymbol{v}}^{\mathrm{zx}}$ are concatenated and fed to two MLPs to predict the SDF value and deformation of the vertex $\boldsymbol{v}$, respectively. Next, a mesh can

be extracted from the DMTet grid using the differentiable Marching Tetrahedra algorithm.

Refinement. We find that meshes extracted from DMTet often bear artifacts as shown in Figure 3. The edge of the extracted mesh is not sharp, and the surface is not smooth: There are evenly distributed tetrahedron-shaped dimples and bumps on the mesh surface. More examples are provided in Figure 5. We also observe similar artifacts in meshes generated by Get3D [8], which uses DMTet to extract meshes as well. It is probably because of the low resolution of the DMTet grid adopted (128^3), and the underlying tetrahedral representation of the DMTet grid. We could try to increase the resolution of the DMTet grid, but the memory and computational cost will increase dramatically, and the topology of the tetrahedral grid could still affect the extracted mesh. Instead, we subdivide and refine the coarse mesh $\boldsymbol{M}$ extracted from DMTet. The architecture of this refinement module is shown in Figure 3. We subdivide each face in the extracted mesh by adding new vertices at the middle point of each edge and connecting them to form 4 smaller faces. The 3D coordinates of vertices of the new mesh $\boldsymbol{M}'$ are fed to a shared MLP to extract features and then projected to a triplane feature $\boldsymbol{h} \in \mathbb{R}^{3\times H\times W\times C}$. We concatenate $\boldsymbol{h}$ with $\boldsymbol{f}$ along the channel dimension and obtain a new triplane feature $[\boldsymbol{f}, \boldsymbol{h}] \in \mathbb{R}^{3\times H\times W\times 2C}$. The concatenated triplane feature $[\boldsymbol{f}, \boldsymbol{h}]$ is fed to a lightweight UNet and we obtain the processed triplane feature $\boldsymbol{h}' \in \mathbb{R}^{3\times H\times W\times C'}$. We refine the subdivided mesh $\boldsymbol{M}'$ by predicting a displacement for each vertex in $\boldsymbol{M}'$ using the triplane feature $\boldsymbol{h}'$. The method is the same as the one that uses $\boldsymbol{f}$ to predict the displacement of each vertex in the DMTet grid. More details of the decoder and refinement module are provided in Appendix Section A.1

Training Loss. The supervision of the autoencoder is added on the latent feature $\boldsymbol{y}$, the upsampled points $\boldsymbol{x}_{\text{u}}$, and the reconstructed mesh $\boldsymbol{M}'$. For $\boldsymbol{y}$, we add a Kullback-Leibler divergence loss between $\boldsymbol{y}$ and the standard Gaussian distribution with weight 10^{-7} in order to make the distribution of latent features relatively simple and smooth. For $\boldsymbol{x}_{\text{u}}$, we first downsample the input point cloud $\boldsymbol{a}$ to the same number of points as $\boldsymbol{x}_{\text{u}}$ using FPS, and then add a Chamfer distance (CD) between the downsampled points and $\boldsymbol{x}_{\text{u}}$ with weight 1. For $\boldsymbol{M}'$, we add a rendering-based loss similar to the one used in [9]. Specifically, $\boldsymbol{M}'$ is fed to a differentiable renderer to obtain the mask silhouette $\boldsymbol{m}$ and depth map $\boldsymbol{d}$, and the rendering-based loss is computed as the sum of L2 distance between $\boldsymbol{m}$ and ground-truth mask silhouette, and L1 distance between $\boldsymbol{d}$ and ground-truth depth map, averaged across N_{view} views. We also warm up the training of the autoencoder by directly supervising the predicted SDF values of the DMTet vertices. See more details in Appendix Section A.1.

3.4 Latent Diffusion Models

After training the autoencoder, we train latent diffusion models on the latent point representation. The representation consists of latent points $\boldsymbol{x} \in \mathbb{R}^{N\times 3}$ and features $\boldsymbol{y} \in \mathbb{R}^{N\times D}$. Note that the number of latent points N could vary between N_{min} and N_{max}. Similar to [21], we train two DDPMs to model the distribution

of $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. The first DDPM $\boldsymbol{\epsilon}_{\text{position}}$ is named position DDPM and learns the distribution of $\boldsymbol{x}$, trained with loss defined in Equation 5. The second DDPM $\boldsymbol{\epsilon}_{\text{feature}}$ is named feature DDPM and learns the distribution of $\boldsymbol{y}$ conditioned on $\boldsymbol{x}$. We use the Transformer architecture in [27] for both $\boldsymbol{\epsilon}_{\text{position}}$ and $\boldsymbol{\epsilon}_{\text{feature}}$. $\boldsymbol{x}$ and $\boldsymbol{y}$ are padded to the maximum length N_{max} during training. See Appendix Section A.2 for more details of the Transformer architecture and training losses.

Sampling. To sample from the trained two DDPMs, we first use $\boldsymbol{\epsilon}_{\text{position}}$ to generate positions $\boldsymbol{x}$ of the latent points. Note that the number of latent points can be chosen arbitrarily between N_{min} and N_{max} by the user during sampling. Then we use $\boldsymbol{\epsilon}_{\text{feature}}$ to generate features $\boldsymbol{y}$ conditioned on $\boldsymbol{x}$. Finally, the generated latent points $\boldsymbol{x}$ with features $\boldsymbol{y}$ can be reconstructed to a mesh by the trained autoencoder.

4 Related Work

Diffusion Models. Diffusion models are likelihood-based generative models composed of a diffusion process and a reverse process. They have been thoroughly explored for image generation [6,10,11,25] and speech synthesis [13,14,31]. To alleviate the slow sampling speed of diffusion models, latent diffusion models [33,37] are proposed to train diffusion models in the latent space of an autoencoder, which encodes data samples to a more compact representation and thus accelerates the training and sampling speed of diffusion models. Our method is based on latent diffusion models.

Diffusion Model for 3D Shape Generation. Diffusion models have also been explored for 3D shape generation. They are first applied to point cloud generation [19, 21, 26, 43, 49]. Some methods [21, 43] propose to reconstruct meshes from the generated point clouds through surface reconstruction techniques [29]. Another line of works [4,9,15,17,24,35,47] utilizes implicit fields [22,28] to generate meshes. They usually design latent representations such as points, voxels, or triplanes to represent the implicit fields, and then train diffusion models on this latent representation. Meshes can be later reconstructed by marching cubes [18] or deep marching tetrahedra (DMTet [34]).

Text-to-Image Model for 3D Shape Generation. Recent works [3, 16, 30, 39, 40] leverage a pre-trained text-to-image diffusion model [33] to generate 3D assets given a text prompt. They first represent a 3D asset as a neural radiance field (NERF [23]) or a mesh with a texture map, then render it as images and apply Score Distillation Sampling (SDS) Loss [30] to optimize the parameters of the 3D asset. However, these methods are quite time-consuming as they need to optimize the parameters of a 3D asset for every single object. It is also difficult for text to accurately control the generated shape.

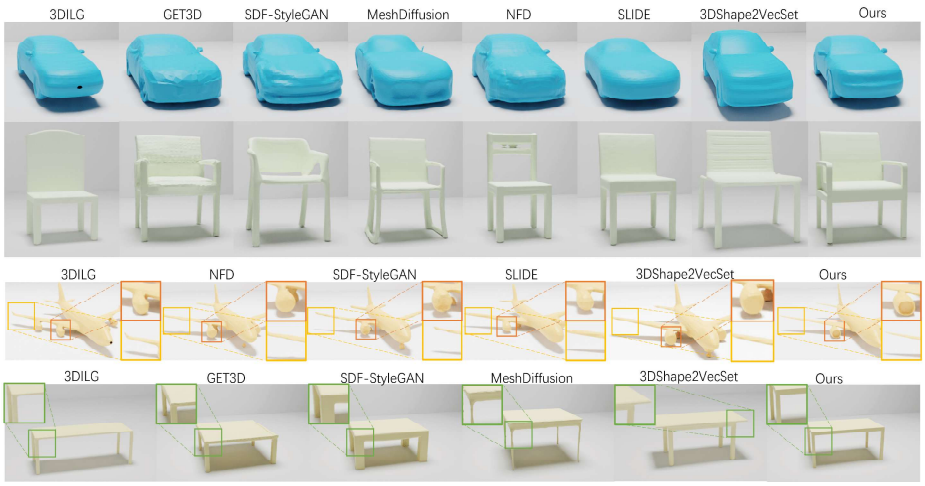

Fig. 4: Visual comparison between meshes generated by our method and baselines. Zoom in to better see the details. More qualitative results are in Appendix Section F.

5 Experiment

5.1 Dataset

We use the ShapeNet [2] dataset to train our model and compare it with baselines. It contains models from 55 categories. We split the dataset into training set (70%), validation set (10%), and test set (20%). We normalize each 3D model in the range of $[-1, 1]^3$. To obtain gound-truth mask silhouettes and depth maps, we render mask silhouettes and depth maps for each 3D model from 100 random views at the resolution of 1024×1024. The radius is fixed at 2.6. Elevation angle and azimuth angle is sampled uniformly from $[90°, -90°]$ and $[0°, 360°]$, respectively. FOV is set to $60°$.

5.2 Implementation Details

Model Architecture. For the mesh autoencoder, we set the number of latent points to $N_{\text{min}} = 128, N_{\text{max}} = 256$. For the point-based encoder, we use 3 layers of SA modules [21]. The size of the triplane is $3 \times 256 \times 256 \times \times 32$. The triplane-based decoder is composed of a UNet with 3D-aware convolutions, a DMTet grid with resolution 128^3, and a refinement module. Detailed architecture of the mesh autoencoder is in Appendix Section A.1. For both the position DDPM and feature DDPM, we use a Transformer architecture composed of 12 Multi-Head Self-Attettion blocks. The embedding dimension of each attention block is 512 and the number of attention heads is 8. Detailed architecture of the position DDPM and feature DDPM is in Appendix Section A.2.

Training. We train the mesh autoencoder on all 55 categories in ShapeNet. We sample $N_{\text{in}} = 16384$ points from the mesh surface as input point cloud. At each training step, we randomly sample $N_{\text{view}} = 8$ views out of the 100 views in the

Table 1: Compare the average generation time per mesh and Shading-FID of our method and baselines. "-" indicates that no checkpoint is provided for that category. To fairly compare the generation speed and quality, we use 1000 denoising steps during inference for all DDPM-based methods, except for MeshDiffusion as it is too slow. We use DDIM [36] (100 Steps) to accelerate MeshDiffusion. MeshDiffusion uses a 128^3 DMTet grid (903.20s per sample) for Car and Chair, and uses a 64^3 DMTet grid (91.35s per sample) for Airplane and Table.

Method	Category	Model Type	Generation Time (s)	Shading-FID ↓			
				Airplane	Car	Chair	Table
SDF-StyleGAN [48]	Single	GAN	1.25	70.49	105.40	46.04	45.57
GET3D [8]	Single	GAN	0.12	-	182.07	66.48	64.06
3DILG [44]	Multiple	Autoregressive	9.91	58.18	152.00	29.71	52.97
NFD [35]	Single	DDPM	7.93	54.43	182.58	42.66	-
MeshDiffusion [17]	Single	DDPM	91.35, 903.20	134.10	151.76	76.81	79.59
SLIDE [21]	Single	DDPM	0.20	85.17	199.84	43.64	-
3DShape2VecSet [45]	Multiple	DDPM	6.56	47.86	**101.38**	22.92	23.90
Ours	Multiple	DDPM	1.17	**32.10**	121.42	**22.16**	**16.50**

Table 2: 1-NNA comparison between our method and baselines. GetMesh achieves the best 1-NNA compared even with single-category models, which demonstrates that meshes generated by GetMesh best match the distribution of the dataset.

Method	Category	1-NNA (CD) (Percent) ↓				1-NNA (EMD) (Percent) ↓			
		Airplane	Car	Chair	Table	Airplane	Car	Chair	Table
SDF-StyleGAN [48]	Single	88.10	95.20	65.22	75.18	91.75	94.65	70.27	75.08
GET3D [8]	Single	-	97.00	68.97	68.37	-	92.90	65.16	67.76
3DILG [44]	Multiple	85.20	95.25	74.27	82.43	88.15	94.30	73.62	81.43
NFD [35]	Single	70.75	86.8	53.5	-	77.3	87.85	55.06	-
MeshDiffusion [17]	Single	73.60	89.20	67.77	57.81	74.60	88.10	67.86	60.21
SLIDE [21]	Single	80.20	91.95	59.41	-	80.25	92.55	61.01	-
3DShape2VecSet [45]	Multiple	72.4	89.45	60.66	56.61	78.25	89.4	60.91	58.66
Ours	Multiple	**69.95**	**84.95**	**52.05**	**52.8**	**69.20**	**81.65**	**53.6**	**52.1**

dataset to supervise the reconstructed mesh. The mesh autoencoder is trained in three phases for 900 epochs with a batchsize of 128. Detailed training schedule and time of the mesh autoencoder are in Appendix Section A.1. For both the position DDPM and feature DDPM, we use the Adam optimizer to train them for 4000 epochs with batchsize 128 and learning rate 2×10^{-4}. More training details of the position DDPM and feature DDPM are in Appendix Section A.2.

5.3 Mesh Autoencoding

The autoencoder is trained on the training set. We select the checkpoint with the best performance on the validation set and evaluate its performance on the test set. We compute the IOU between rendered masks of the reconstructed meshes and ground-truth masks (averaged across the first 32 views in the dataset), and L_2 CD (Chamfer Distance) loss between 10^5 surface points sampled from the reconstructed meshes and ground-truth meshes. Our mesh autoencoder achieves 0.968 IOU and 1.34×10^{-3} CD loss.

Table 3: MMD comparison between our method and baselines. GetMesh achieves good MMD compared even with single-category models, which demonstrates the high generation quality of GetMesh.

Method	Category	MMD (CD) ×1000 ↓				MMD (EMD) ×100 ↓			
		Airplane	Car	Chair	Table	Airplane	Car	Chair	Table
SDF-StyleGAN [48]	Single	4.31	4.87	16.42	23.05	11.44	8.82	16.95	17.17
GET3D [8]	Single	-	4.97	17.16	21.08	-	8.67	16.61	16.57
3DILG [44]	Multiple	4.49	5.18	18.56	30.13	9.33	8.80	17.34	19.39
NFD [35]	Single	3.08	4.17	**14.44**	-	8.38	8.15	**15.00**	-
MeshDiffusion [17]	Single	**3.02**	4.79	17.11	18.68	**8.10**	8.88	17.17	15.82
SLIDE [21]	Single	3.60	4.43	15.03	-	8.64	8.38	15.88	-
3DShape2VecSet [45]	Multiple	3.16	4.20	15.51	18.69	8.79	8.41	16.20	15.88
Ours	Multiple	3.38	**4.01**	14.55	**17.86**	8.31	**8.00**	15.72	**15.34**

Table 4: Coverage comparison between our method and baselines. GetMesh achieves high Coverage compared even with single-category models, which demonstrates the high generation diversity of GetMesh.

Method	Category	Coverage (CD) (Percent) ↑				Coverage (EMD) (Percent) ↑			
		Airplane	Car	Chair	Table	Airplane	Car	Chair	Table
SDF-StyleGAN [48]	Single	32.5	20.0	43.24	38.13	21.7	19.8	38.43	35.83
GET3D [8]	Single	-	20.5	45.95	44.04	-	23.4	47.75	45.95
3DILG [44]	Multiple	35.5	13.6	36.54	24.22	28.3	13.4	37.84	27.23
NFD [35]	Single	46.4	**29.5**	46.75	-	36.8	27.0	46.65	-
MeshDiffusion [17]	Single	**48.0**	25.2	40.14	49.05	43.8	28.3	42.54	45.25
SLIDE [21]	Single	40.8	26.3	46.75	-	38.1	23.7	46.65	-
3DShape2VecSet [45]	Multiple	47.0	28.4	**50.45**	49.95	43.1	27.6	49.35	50.25
Ours	Multiple	46.1	29.0	49.35	**51.45**	**50.5**	**30.4**	**50.35**	**51.15**

5.4 Mesh Generation

We train the position DDPM and the feature DDPM in the latent space of the pre-trained autoencoder in the previous section. We train class-conditional DDPMs on the 55 categories. Each class is associated with a learnable token and the token is appended to the sequence of positions and features of the latent points during training. We randomly drop the class label to null with 20% probability during training to enable classifier-free [12] guidance in the sampling phase, and novel shape generation and editing beyond the 55 categories in ShapeNet.

We compare our method with both multi-category generative models [44, 45] and single-category generative models [8, 17, 21, 35, 48]. We use Shading FID [48], 1-NNA [42], Minimum Matching Distance (MMD [1]) and Coverage [1] to evaluate our method and baselines. More evaluation details are in Appendix Section C. Results are shown in Table 1, Table 2, Table 3 and Table 4, respectively. We can see that our model achieves the best 1-NNA in all cases and the best Shading-FID in most cases even compared with single-category models. Both metrics measure the distance between the distribution of the generated meshes and meshes in the reference dataset. Therefore, the experiments demonstrate that GetMesh better learns the distribution of meshes in the dataset compared with baselines. MMD and Coverage explicitly measure the quality and diversity of generated meshes,

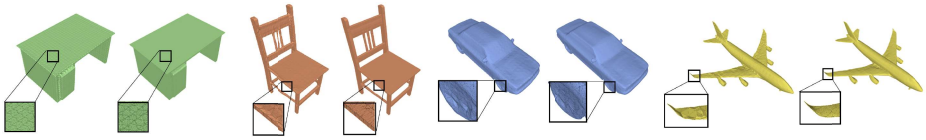

Fig. 5: Compare meshes reconstructed by autoencoders with and without the refinement module. For each pair of meshes, the left one is without the refinement module, and the right one is with the module. Zoom in to see more details.

Table 5: Ablation study of the number of latent points in the latent space. Shading FID is reported.

N_{min}	N_{max}	Airplane	Car	Chair	Table
32	64	206.57	261.49	224.71	228.43
128	256	**32.10**	121.42	**24.59**	**16.50**
512	1024	44.32	**113.22**	63.25	36.03
2048	4096	237.36	163.16	162.83	117.55

respectively. We can see that GetMesh achieves highly competitive MMD and Coverage even compared with single-category models, which demonstrates the superior generation quality and diversity of GetMesh.

Besides quantitative comparisons, we also qualitatively compare meshes generated by GetMesh and baselines in Figure 4. We can see that our method generates meshes with sharper edges and smoother surfaces. Our method also generates thin structures such as airplane wings and tails quite well.

Generation speed. We compare the average generation time per sample of our method and baselines tested on a single NVIDIA A100 GPU in Table 1. We can see that GetMesh is much faster than other DDPM-based method. That is because we train diffusion models on a much more compact point-based representation, which contains $128 \sim 256$ points. Note that triplanes are only used in our autoencoder to reconstruct high-quality meshes, not used to train diffusion models. On the other hand, MeshDiffusion trains diffusion models on the DMTet grid, which contains 277410 vertices for a 128^3 resolution grid. NFD trains diffusion models on triplanes of resolution 128, which contains $128 \times 128 \times 3 = 49152$ pixels. 3DShape2VecSet trains diffusion models on a set of 512 vectors. This explains why GetMesh is much faster than these DDPM-based methods. Although SLIDE [21] is faster than our method by using fewer points (16 points), GetMesh enables much more accurate control over the generated shapes than SLIDE by using relatively more points as shown in Figure 6. In addition, GetMesh generates much higher quality meshes than SLIDE by using a triplane-based mesh decoder compared to SLIDE's Poisson-based surface reconstruction method [29]. The generation speed of GetMesh is also competitive compared with other non-DDPM-based methods.

5.5 Ablation Study

We conduct an ablation study on the number of latent points in the latent space. The performance of the mesh autoencoders with different numbers of

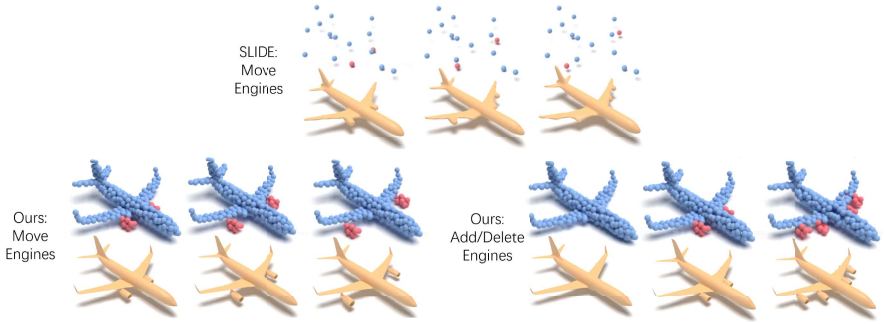

Fig. 6: Compare our method with SLIDE in terms of controllable generation. Our method can perform more delicate control on the generated shape and enables adding or deleting shape parts.

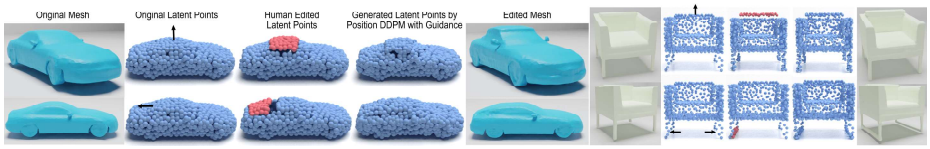

Fig. 7: Human-edited latent points could have holes, missing parts, or other flaws. We develop a guidance method for the position DDPM to generate latent points that are consistent with the edited latent points, but mitigate flaws in the edited latent points. This guidance method makes mesh manipulation through latent point positions more robust and convenient.

latent points is shown in Appendix Section D. The mesh autoencoder with $N \in [512, 1024]$ achieves the best reconstruction performance. Next, we train latent diffusion models in the latent space of these autoencoders. We report the generation performance of the latent diffusion models in Table 5 and more results are in Appendix Section D. We can see that the diffusion model with $N \in [128, 256]$ achieves the best generation performance in most cases, and the diffusion model with $N \in [512, 1024]$ achieves relatively good generation performance as well. On the other hand, the performance of the diffusion models with too many ($N \in [2048, 4096]$) or too few ($N \in [32, 64]$) latent points is significantly worse than the other two diffusion models. We need a moderate number of latent points ($128 \sim 1024$) to train latent diffusion models with good generation performance.

Next, we ablate our refinement module in the mesh autoencoder. Without the refinement module, the reconstruction performance of the mesh autoencoder ($N \in [128, 256]$) drops from 0.968 IOU and 1.34×10^{-3} CD loss to 0.961 IOU and 1.46×10^{-3} CD loss. Figure 5 also provides a qualitative comparison between meshes reconstructed with and without the refinement module. We can see that the refinement module greatly mitigates the artifacts in the coarse meshes extracted from DMTet. It makes the mesh surface smoother and the edges sharper.

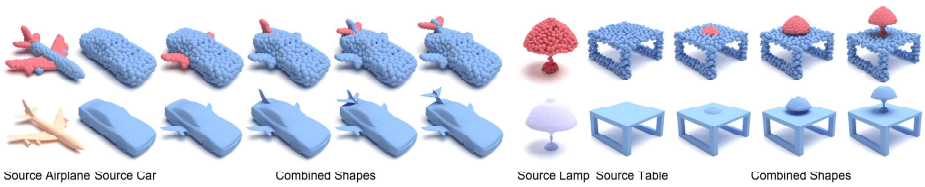

Fig. 8: We can combine different shapes to form novel shapes by combining their latent points. Our models can properly handle the joints of different parts and output watertight meshes.

Fig. 9: Shape interpolation.

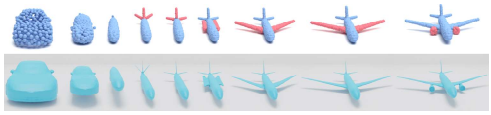

Fig. 10: Turning a car into an airplane.

5.6 Controllable Mesh Generation

Our method enables highly controllable 3D shape generation and flexible mesh manipulation by adjusting the number, positions, or features of the latent points. We demonstrate the controllable generation ability of the model with $N \in [512, 1024]$ in this section, and more examples of the model with $N \in [128, 256]$ are shown in Appendix Section E.

Controllable mesh generation. We can use the positions of the latent points to control the shape of generated meshes. We compare our method with SLIDE [21], which also supports latent point-based shape manipulation. Our models have more latent points than SLIDE (16 points), therefore, we can perform more delicate controls over the generated mesh. As shown in Figure 6, we can control the positions of the engines of the generated airplane while SLIDE struggles to control. In addition, our models support varying numbers of latent points, and thus we can delete or add a part of a shape. Figure 6 shows that we can delete or add engines to the airplane, which SLIDE can not achieve since it uses a fixed number of latent points.

We also develop a guided-sampling method for the position DDPM to facilitate controllable mesh generation and manipulation. Human-edited latent points may have flaws in many cases. Therefore, we need to develop a method that reflects the editing effect of a user's intention, while mitigating the pitfalls of human editing. The main idea of our method is to leverage the position DDPM's ability to generate an arbitrary number of latent points and use the edited latent points to guide the sampling process of the position DDPM. The guidance method is inspired by the one proposed in [7] and is explained in Appendix Section B. In Figure 7, we demonstrate that our guidance method can make the position DDPM generate latent points that are consistent with the edited latent points but mitigate their flaws, and thus facilitate the process of shape manipulation.

Shape combination. Since our models support varying numbers of latent points, we can directly combine different shapes together to form novel shapes. We can combine the latent points and regenerate features using the feature

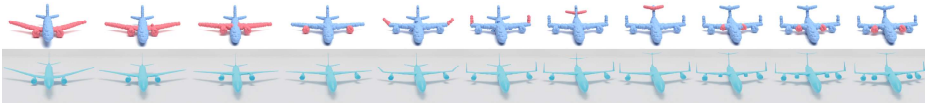

Fig. 11: Turning a twin-engine passenger airplane into a four-engine transport airplane.

Fig. 12: We use MATLABER [41] to generate materials for meshes generated by our method. More examples are in Appendix Section F. Note that texture or material generation is NOT the focus or contribution of this work. We merely intend to show that our method can be seamlessly combined with off-the-shelf methods to obtain meshes with textures or materials.

DDPM or simply remaining the original features, and then use the mesh autoencoder to decode the combined latent points to a mesh. Figure 8 shows that our models can properly handle the joints of different parts.

Shape interpolation. Shape interpolation is straightforward using latent points and their features. For two shapes represented by latent points $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{R}^{N\times 3}$ and latent features $\boldsymbol{y}_1, \boldsymbol{y}_2 \in \mathbb{R}^{N\times D}$, we find the optimal bijection $\phi^* : \boldsymbol{x}_1 \longrightarrow \boldsymbol{x}_2$.

$$\phi^* = \underset{\phi}{\operatorname{argmin}} \, \Sigma_{i=1}^{N} ||\boldsymbol{x}_1^i - \phi(\boldsymbol{x}_1^i)||_2, \tag{6}$$

where $\boldsymbol{x}_1^i$ is the i-th point in $\boldsymbol{x}_1$. Based on this bijection ϕ^*, we can interpolate the corresponding points and their features between $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. Figure 9 gives an example of shape interpolation.

Shape animation. Traditional mesh animation methods drive 3D meshes by deforming the mesh. The vertices of the mesh are deformed while the topology of the mesh is fixed, namely, the connections of the vertices are fixed. This could prevent them from performing some complex shape animations that require topology changes. On the other hand, our method can generate meshes of arbitrary topology given the latent points. Therefore, we can use the latent points to drive the mesh. Specifically, we can use a sequence of latent point positions to create an animation involving complex shape transformations. Figure 11 and Figure 10 are two examples. The complete video can be found in the supplementary material. We think that our method provides a complementary approach to existing mesh animation methods and enables more creative and complex animations of 3D meshes.

Textured mesh generation. It is possible to paint our generated meshes with some off-the-shelf methods. We use MATLABER [41] to colorize our generated meshes, which can generate materials for meshes by leveraging a powerful text-to-image diffusion model. Some examples are shown in Figure 12, and more examples are in Appendix Section F.

6 Limitations

While GetMesh has made significant progress in 3D generation quality and controllability, it still has some limitations. Firstly, training GetMesh requires ground-truth 3D data, which is quite expensive to acquire compared with 2D images. In light of recent works [3, 16, 30, 39, 40] that generate 3D assets using 2D Text-to-Image diffusion models trained on large-scale image datasets, it is possible to combine the 3D priors in GetMesh with 2D priors in Text-to-Image diffusion models to achieve more diverse and high-quality 3D generation. In addition, GetMesh is only validated on the ShapeNet [2] dataset due to limited computational resources. We plan to further verify the scalability of GetMesh on larger-scale datasets such as Objaverse [5] in the future.

7 Conclusion

In this paper, we propose `GetMesh`, a multi-category generative model that enables both high-quality mesh generation and flexible control over the generated shape. It combines the advantages of both point-based representation and triplane-based representation. The triplane-based representation associated with a decoder and a refinement module enables us to reconstruct high-quality meshes from the latent representation. By adjusting the number, positions or features of the latent points, we can intuitively and robustly change global/local topologies of meshes, add/remove mesh parts, as well as combine mesh parts across different instances/categories.

References

1. Achlioptas, P., Diamanti, O., Mitliagkas, I., Guibas, L.: Learning representations and generative models for 3d point clouds. In: International conference on machine learning. pp. 40–49. PMLR (2018) 10
2. Chang, A.X., Funkhouser, T., Guibas, L., Hanrahan, P., Huang, Q., Li, Z., Savarese, S., Savva, M., Song, S., Su, H., et al.: Shapenet: An information-rich 3d model repository. arXiv preprint arXiv:1512.03012 (2015) 2, 8, 15
3. Chen, R., Chen, Y., Jiao, N., Jia, K.: Fantasia3d: Disentangling geometry and appearance for high-quality text-to-3d content creation. ArXiv **abs/2303.13873** (2023), https://api.semanticscholar.org/CorpusID:257757213 2, 7, 15
4. Chou, G., Bahat, Y., Heide, F.: Diffusionsdf: Conditional generative modeling of signed distance functions. ArXiv **abs/2211.13757** (2022), https://api.semanticscholar.org/CorpusID:254017862 7
5. Deitke, M., Schwenk, D., Salvador, J., Weihs, L., Michel, O., VanderBilt, E., Schmidt, L., Ehsani, K., Kembhavi, A., Farhadi, A.: Objaverse: A universe of annotated 3d objects. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 13142–13153 (2023) 15
6. Dhariwal, P., Nichol, A.: Diffusion models beat gans on image synthesis. ArXiv **abs/2105.05233** (2021), https://api.semanticscholar.org/CorpusID:234357997 7

7. Fei, B., Lyu, Z., Pan, L., Zhang, J., Yang, W., Luo, T., Zhang, B., Dai, B.: Generative diffusion prior for unified image restoration and enhancement. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 9935–9946 (2023) 13
8. Gao, J., Shen, T., Wang, Z., Chen, W., Yin, K., Li, D., Litany, O., Gojcic, Z., Fidler, S.: Get3d: A generative model of high quality 3d textured shapes learned from images. ArXiv **abs/2209.11163** (2022), https://api.semanticscholar.org/CorpusID:252438648 2, 5, 6, 9, 10
9. Gupta, A., Xiong, W., Nie, Y., Jones, I., Oğuz, B.: 3dgen: Triplane latent diffusion for textured mesh generation. arXiv preprint arXiv:2303.05371 (2023) 2, 5, 6, 7
10. Ho, J., Jain, A., Abbeel, P.: Denoising diffusion probabilistic models. Advances in neural information processing systems **33**, 6840–6851 (2020) 3, 7
11. Ho, J., Saharia, C., Chan, W., Fleet, D.J., Norouzi, M., Salimans, T.: Cascaded diffusion models for high fidelity image generation. J. Mach. Learn. Res. **23**, 47:1–47:33 (2021), https://api.semanticscholar.org/CorpusID:235619773 7
12. Ho, J., Salimans, T.: Classifier-free diffusion guidance. arXiv preprint arXiv:2207.12598 (2022) 10
13. Jeong, M., Kim, H., Cheon, S.J., Choi, B.J., Kim, N.S.: Diff-tts: A denoising diffusion model for text-to-speech. In: Interspeech (2021), https://api.semanticscholar.org/CorpusID:233025015 7
14. Kong, Z., Ping, W., Huang, J., Zhao, K., Catanzaro, B.: Diffwave: A versatile diffusion model for audio synthesis. ArXiv **abs/2009.09761** (2020), https://api.semanticscholar.org/CorpusID:221818900 7
15. Li, M., Duan, Y., Zhou, J., Lu, J.: Diffusion-sdf: Text-to-shape via voxelized diffusion. 2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) pp. 12642–12651 (2022), https://api.semanticscholar.org/CorpusID:254366593 2, 7
16. Lin, C.H., Gao, J., Tang, L., Takikawa, T., Zeng, X., Huang, X., Kreis, K., Fidler, S., Liu, M.Y., Lin, T.Y.: Magic3d: High-resolution text-to-3d content creation. 2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) pp. 300–309 (2022), https://api.semanticscholar.org/CorpusID:253708074 2, 7, 15
17. Liu, Z., Feng, Y., Black, M.J., Nowrouzezahrai, D., Paull, L., yu Liu, W.: Meshdiffusion: Score-based generative 3d mesh modeling. ArXiv **abs/2303.08133** (2023), https://api.semanticscholar.org/CorpusID:257505014 2, 7, 9, 10
18. Lorensen, W.E., Cline, H.E.: Marching cubes: A high resolution 3d surface construction algorithm. Proceedings of the 14th annual conference on Computer graphics and interactive techniques (1987), https://api.semanticscholar.org/CorpusID:15545924 7
19. Luo, S., Hu, W.: Diffusion probabilistic models for 3d point cloud generation. 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) pp. 2836–2844 (2021), https://api.semanticscholar.org/CorpusID:232092778 7
20. Lyu, Z., Kong, Z., Xu, X., Pan, L., Lin, D.: A conditional point diffusion-refinement paradigm for 3d point cloud completion. arXiv preprint arXiv:2112.03530 (2021) 4
21. Lyu, Z., Wang, J., An, Y., Zhang, Y., Lin, D., Dai, B.: Controllable mesh generation through sparse latent point diffusion models. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 271–280 (2023) 1, 4, 6, 7, 8, 9, 10, 11, 13

22. Mescheder, L.M., Oechsle, M., Niemeyer, M., Nowozin, S., Geiger, A.: Occupancy networks: Learning 3d reconstruction in function space. 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) pp. 4455–4465 (2018), https://api.semanticscholar.org/CorpusID:54465161 7
23. Mildenhall, B., Srinivasan, P.P., Tancik, M., Barron, J.T., Ramamoorthi, R., Ng, R.: Nerf: Representing scenes as neural radiance fields for view synthesis. Commun. ACM **65**(1), 99–106 (dec 2021). https://doi.org/10.1145/3503250, https://doi.org/10.1145/3503250 7
24. Nam, G., Khlifi, M., Rodriguez, A., Tono, A., Zhou, L., Guerrero, P.: 3d-ldm: Neural implicit 3d shape generation with latent diffusion models. ArXiv **abs/2212.00842** (2022), https://api.semanticscholar.org/CorpusID:254220714 7
25. Nichol, A., Dhariwal, P.: Improved denoising diffusion probabilistic models. ArXiv **abs/2102.09672** (2021), https://api.semanticscholar.org/CorpusID:231979499 7
26. Nichol, A., Jun, H., Dhariwal, P., Mishkin, P., Chen, M.: Point-e: A system for generating 3d point clouds from complex prompts. ArXiv **abs/2212.08751** (2022), https://api.semanticscholar.org/CorpusID:254854214 7
27. Pang, Y., Wang, W., Tay, F.E., Liu, W., Tian, Y., Yuan, L.: Masked autoencoders for point cloud self-supervised learning. In: European conference on computer vision. pp. 604–621. Springer (2022) 7, 3
28. Park, J.J., Florence, P.R., Straub, J., Newcombe, R.A., Lovegrove, S.: Deepsdf: Learning continuous signed distance functions for shape representation. 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) pp. 165–174 (2019), https://api.semanticscholar.org/CorpusID:58007025 7
29. Peng, S., Jiang, C.M., Liao, Y., Niemeyer, M., Pollefeys, M., Geiger, A.: Shape as points: A differentiable poisson solver. In: Neural Information Processing Systems (2021), https://api.semanticscholar.org/CorpusID:235358422 1, 7, 11
30. Poole, B., Jain, A., Barron, J.T., Mildenhall, B.: Dreamfusion: Text-to-3d using 2d diffusion. ArXiv **abs/2209.14988** (2022), https://api.semanticscholar.org/CorpusID:252596091 2, 7, 15
31. Popov, V., Vovk, I., Gogoryan, V., Sadekova, T., Kudinov, M.A.: Grad-tts: A diffusion probabilistic model for text-to-speech. In: International Conference on Machine Learning (2021), https://api.semanticscholar.org/CorpusID:234483016 7
32. Qi, C.R., Yi, L., Su, H., Guibas, L.J.: Pointnet++: Deep hierarchical feature learning on point sets in a metric space. Advances in neural information processing systems **30** (2017) 4
33. Rombach, R., Blattmann, A., Lorenz, D., Esser, P., Ommer, B.: High-resolution image synthesis with latent diffusion models. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 10684–10695 (2022) 4, 7
34. Shen, T., Gao, J., Yin, K., Liu, M.Y., Fidler, S.: Deep marching tetrahedra: a hybrid representation for high-resolution 3d shape synthesis. Advances in Neural Information Processing Systems **34**, 6087–6101 (2021) 5, 7
35. Shue, J., Chan, E., Po, R., Ankner, Z., Wu, J., Wetzstein, G.: 3d neural field generation using triplane diffusion. 2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) pp. 20875–20886 (2022), https://api.semanticscholar.org/CorpusID:254095843 2, 5, 7, 9, 10
36. Song, J., Meng, C., Ermon, S.: Denoising diffusion implicit models. arXiv preprint arXiv:2010.02502 (2020) 9

37. Vahdat, A., Kreis, K., Kautz, J.: Score-based generative modeling in latent space. Advances in Neural Information Processing Systems **34**, 11287–11302 (2021) 4, 7
38. Wang, T., Zhang, B., Zhang, T., Gu, S., Bao, J., Baltrusaitis, T., Shen, J., Chen, D., Wen, F., Chen, Q., et al.: Rodin: A generative model for sculpting 3d digital avatars using diffusion. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 4563–4573 (2023) 5
39. Wang, Z., Lu, C., Wang, Y., Bao, F., Li, C., Su, H., Zhu, J.: Prolificdreamer: High-fidelity and diverse text-to-3d generation with variational score distillation. ArXiv **abs/2305.16213** (2023), https://api.semanticscholar.org/CorpusID:258887357 2, 7, 15
40. Xu, J., Wang, X., Cheng, W., Cao, Y.P., Shan, Y., Qie, X., Gao, S.: Dream3d: Zero-shot text-to-3d synthesis using 3d shape prior and text-to-image diffusion models. 2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) pp. 20908–20918 (2022), https://api.semanticscholar.org/CorpusID:255340806 2, 7, 15
41. Xu, X., Lyu, Z., Pan, X., Dai, B.: Matlaber: Material-aware text-to-3d via latent brdf auto-encoder. arXiv preprint arXiv:2308.09278 (2023) 14, 11
42. Yang, G., Huang, X., Hao, Z., Liu, M.Y., Belongie, S.J., Hariharan, B.: Pointflow: 3d point cloud generation with continuous normalizing flows. 2019 IEEE/CVF International Conference on Computer Vision (ICCV) pp. 4540–4549 (2019), https://api.semanticscholar.org/CorpusID:195750453 10, 6
43. Zeng, X., Vahdat, A., Williams, F., Gojcic, Z., Litany, O., Fidler, S., Kreis, K.: Lion: Latent point diffusion models for 3d shape generation. ArXiv **abs/2210.06978** (2022), https://api.semanticscholar.org/CorpusID:252872881 1, 7
44. Zhang, B., Nießner, M., Wonka, P.: 3dilg: Irregular latent grids for 3d generative modeling. ArXiv **abs/2205.13914** (2022), https://api.semanticscholar.org/CorpusID:249152155 9, 10
45. Zhang, B., Tang, J., Niessner, M., Wonka, P.: 3dshape2vecset: A 3d shape representation for neural fields and generative diffusion models. arXiv preprint arXiv:2301.11445 (2023) 9, 10
46. Zheng, X., Liu, Y., Wang, P.S., Tong, X.: Sdf-stylegan: Implicit sdf-based stylegan for 3d shape generation. Computer Graphics Forum **41** (2022), https://api.semanticscholar.org/CorpusID:250048592 2
47. Zheng, X., Pan, H., Wang, P.S., Tong, X., Liu, Y., yeung Shum, H.: Locally attentional sdf diffusion for controllable 3d shape generation. ACM Transactions on Graphics (TOG) **42**, 1 – 13 (2023), https://api.semanticscholar.org/CorpusID:258557967 2, 7
48. Zheng, X., Liu, Y., Wang, P., Tong, X.: Sdf-stylegan: Implicit sdf-based stylegan for 3d shape generation. In: Computer Graphics Forum. vol. 41, pp. 52–63. Wiley Online Library (2022) 9, 10, 5
49. Zhou, L., Du, Y., Wu, J.: 3d shape generation and completion through point-voxel diffusion. 2021 IEEE/CVF International Conference on Computer Vision (ICCV) pp. 5806–5815 (2021), https://api.semanticscholar.org/CorpusID:233182041 7

A Network Architectures and Training Details

In this section, we explain the detailed architecture and training procedure of our mesh autoencoder, position DDPM, and feature DDPM. We will release the code to facilitate reproducibility if the paper is accepted.

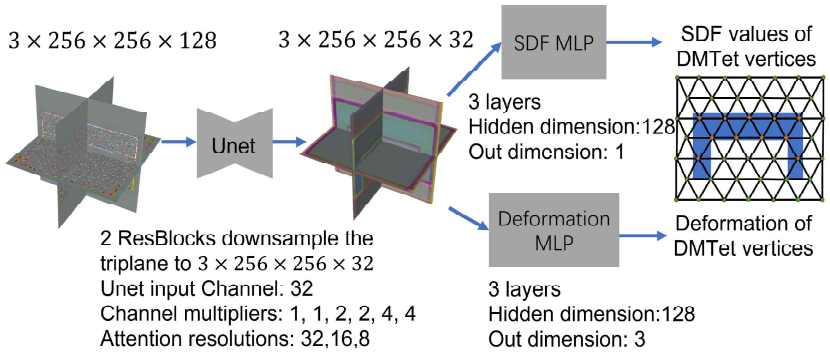

Fig. 13: Detailed architecture of the triplane-based decoder.

A.1 Mesh Autoencoder

The mesh autoencoder is composed of the point-based encoder, the triplane-based decoder, and the refinement module. The point-based encoder is similar to the one proposed in [21]. It gradually downsamples the input point cloud and propagates features level by level. The detailed architecture of the point-based encoder is shown in Table 7. Recall that we need to re-organize the latent point representation to the triplane representation. We first upsample the latent points and then project them to the triplane. The details of the upsampling process are shown in Table 7. The detailed architecture of the triplane-based decoder is shown in Figure 13, and the detailed architecture of the refinement module is shown in Figure 14.

To handle varying numbers of latent points $N \in [N_{\min}, N_{\max}]$ in the mesh autoencoder, we pad the latent points to $N_{\max}$ points with dummy points at 3D coordinate $(4, 4, 4)$, which are far away from the object bounding box $[-1, 1]^3$, such that the dummy points will not affect features propagated to the real latent points. The upsampling process is based on the one proposed in [21]. In the upsampling process, each real latent point is split into γ new points by using its feature to predict γ offsets through a shared MLP. The resulting points that range from $\gamma N_{\min}$ to $\gamma N_{\max}$ points are downsampled to a fixed number of points by furthest point sampling. We use $\gamma = 16$ in all experiments.

Training details. The supervision of the autoencoder is added on the latent feature $\boldsymbol{y}$, the upsampled points $\boldsymbol{x}_{\mathrm{u}}$, the reconstructed mesh $\boldsymbol{M}'$, and the predicted

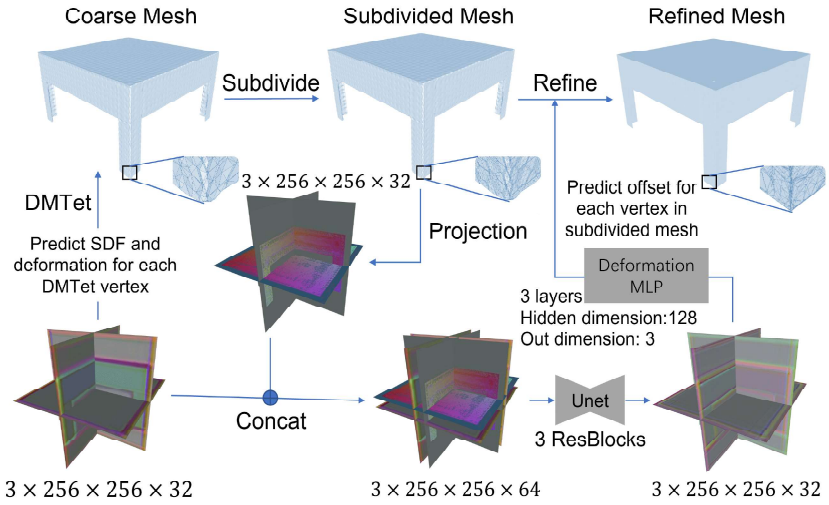

Fig. 14: Detailed architecture of the refinement module.

Table 6: The schedule of L_{SDF} and L_{Render} to train the mesh autoencoder.

Epoch	300	320	325	330	335	340	345	350	355	360	365	600
L_{SDF}	1	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1	0	0
L_{Render}	0	0.001	0.002	0.005	0.01	0.02	0.05	0.1	0.2	0.5	1	1

SDF values of the DMTet grid. For $\boldsymbol{y}$, we add a Kullback-Leibler divergence loss, L_{KL}, between $\boldsymbol{y}$ and the standard Gaussian distribution in order to make the distribution of latent features relatively simple and smooth. For $\boldsymbol{x}_{\text{u}}$, we first downsample the input point cloud $\boldsymbol{a}$ to the same number of points as $\boldsymbol{x}_{\text{u}}$ using FPS, and then add a Chamfer distance (CD), L_{CD}, between the downsampled points and $\boldsymbol{x}_{\text{u}}$. For $\boldsymbol{M}'$, we add a rendering-based loss, L_{Render}, similar to the one used in [9]. Specifically, $\boldsymbol{M}'$ is fed to a differentiable renderer to obtain the mask silhouette $\boldsymbol{m}$ and depth map $\boldsymbol{d}$, and the rendering-based loss is computed as the sum of L2 distance between $\boldsymbol{m}$ and ground-truth mask silhouette, and L1 distance between $\boldsymbol{d}$ and ground-truth depth map, averaged across N_{view} views. For the predicted SDF values of the DMTet grid, we add an MSE loss, L_{SDF}, between the predicted SDF values and ground-truth SDF values of the DMTet grid.

The autoencoder is trained in three phases. In all phases, we use a weight of 10^{-7} for L_{KL} and use a batchsize of 128. In the first phase, we use a weight of 1 for L_{CD}, and the other loss terms are not included. The triplane-based decoder is not trained in this phase. We use the Adam optimizer with a learning rate of 10^{-3} and train the mesh autoencoder for 300 epochs. The checkpoint with the lowest L_{CD} is selected to train the mesh autoencoder in the second phase.

Algorithm 1: Guided sampling for position DDPM.

Input: Trained position DDPM $\boldsymbol{\epsilon}_{\text{position}}$. Edited latent point positions $\boldsymbol{x}_e \in \mathbb{R}^{N_e \times 3}$. Guidance scale s.
Output: Sampled latent points $\boldsymbol{x}^0$.
Sample $\boldsymbol{x}^T$ from $\mathcal{N}(\boldsymbol{x}^T; 0, \boldsymbol{I})$
for t *from* T *to* 1 **do**
  $\tilde{\boldsymbol{x}}^0 = \frac{\boldsymbol{x}^t}{\sqrt{\bar{\alpha}_t}} - \frac{\sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_{\text{position}}(\boldsymbol{x}^t, t)}{\sqrt{\bar{\alpha}_t}}$
  $\mathcal{L} = \|\tilde{\boldsymbol{x}}^0[0 : N_e, :] - \boldsymbol{x}_e\|^2$
  $\tilde{\boldsymbol{x}}^0 \leftarrow \tilde{\boldsymbol{x}}^0 - s\nabla_{\tilde{\boldsymbol{x}}^0}\mathcal{L}$
  $\tilde{\boldsymbol{\mu}}_t = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\tilde{\boldsymbol{x}}^0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\boldsymbol{x}^t$
  $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$
  Sample $\boldsymbol{x}^{t-1}$ from $\mathcal{N}(\boldsymbol{x}^{t-1}; \tilde{\boldsymbol{\mu}}_t, \tilde{\beta}_t\boldsymbol{I})$
end
return $\boldsymbol{x}^0$

In the second phase, L_{CD} is kept with weight 1. We add a warm-up schedule for L_{Render} and L_{SDF} to ensure that DMTet can extract meaningful meshes at the initial phase. The detailed schedule is shown in Table 6. In the second phase, we do not use the refinement module and L_{Render} is applied to the coarse mesh extracted from DMTet. We use the Adam optimizer with a learning rate of 10^{-3} and train the mesh autoencoder for 300 epochs. The checkpoint with the lowest L_{Render} is selected to train the mesh autoencoder in the third phase.

In the third phase, we use L_{CD} with weight 1 and L_{Render} with weight 1. The refinement module is included and L_{Render} is applied to the refined mesh. We use the Adam optimizer with a learning rate of 5×10^{-4} and train the mesh autoencoder for 300 epochs. The checkpoint with the lowest L_{Render} is selected to train the latent diffusion models. It takes about a week to train the mesh autoencoder on 32 NVIDIA A100 GPUs in the three phases.

A.2 Latent Diffusion Models

For the position DDPMs and feature DDPMs in Section 3.4, we use the Transformer architecture proposed for points in [27]. They all share the same architecture: The Transformer is composed of 12 Multi-Head Self-Attettion blocks. The embedding dimension of each attention block is 512 and the number of attention heads is 8. The class label is mapped to a learnable 512-dimension embedding and appended to the input sequence.

For the position DDPM, the 3D coordinates of the latent points are mapped to 512-dimension embeddings through a shared MLP, and the 512-dimension embeddings are treated as both inputs and positional embeddings to the Transformer. The Transformer outputs are fed to a shared MLP to predict the noises added to the latent point 3D coordinates. For the feature DDPM, both 3D coordinates and features of the latent points are fed to the Transformer. The 3D coordinates are mapped to 512-dimension embeddings through a shared MLP

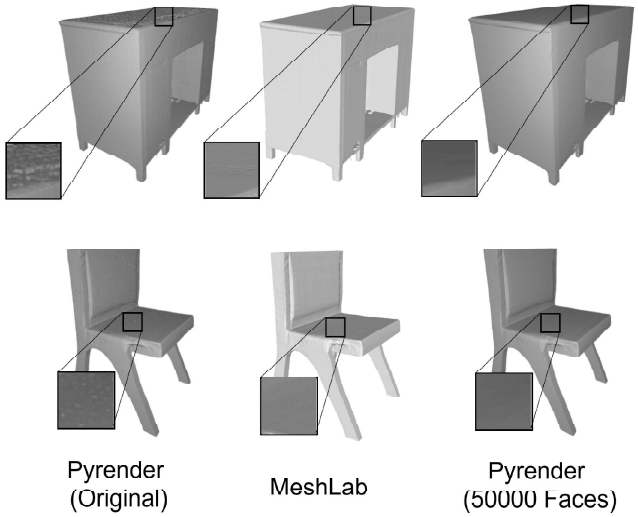

Fig. 15: We use Open3D's simplify-quadric-decimation to reduce the number of faces in an object to 50000 in order to alleviate the artifacts from Pyrender.

and are treated as positional embeddings. The features are mapped to 512-dimension embeddings through another shared MLP and are treated as inputs to the Transformer. The Transformer outputs are fed to a shared MLP to predict the noises added to the features of the latent points.

Traing details. The position and feature DDPMs are trained for 4000 epochs with batchsize 128 and learning rate 2×10^{-4}. The Adam optimizer is used. The amount of computational resources and time to train them is shown in Table 9.

B Guided Sampling of Position DDPM

We develop a guided-sampling method for the position DDPM to facilitate controllable mesh generation and editing. Human-edited latent points may have flaws in many cases. Therefore, we need to develop a method that reflects the editing effect of a user's intention, while mitigating the pitfalls of human editing. The main idea of our method is to leverage the position DDPM's ability to generate an arbitrary number of latent points and add guidance to the sampling process of the position DDPM.

Assume human-edited latent point is $\boldsymbol{x}_e \in \mathbb{R}^{N_e \times 3}$. We aim to generate latent points $\boldsymbol{x}^0 \in \mathbb{R}^{N \times 3}$ that reflects changes in $\boldsymbol{x}_e$, while avoids flaws in $\boldsymbol{x}_e$. We assume $N_e < N$. This can always be attained by setting $N = N_{\max}$ and reducing the number of points in $\boldsymbol{x}_e$ using FPS. Next, we use $\boldsymbol{x}_e$ to guide the sampling process of the position DDPM, and the detailed algorithm is shown in Algorithm 1. We set the guidance scale s to 1 in our experiments. The overall idea of the algorithm is to guide the first N_e points in the $N_{\max}$ generated points and encourage them

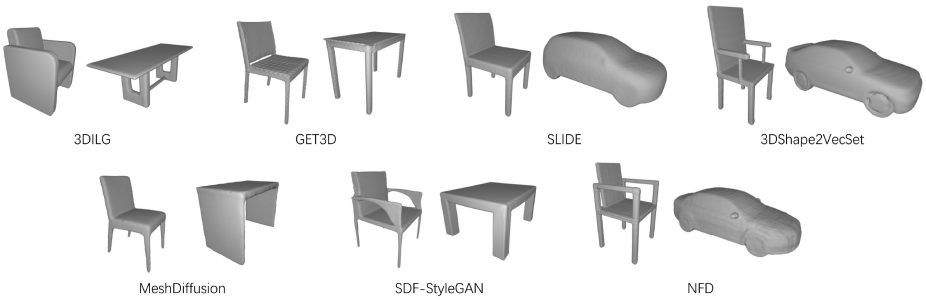

Fig. 16: Pyrender-rendered images of meshes generated by baseline methods.

to be close to the human-edited latent points. The rest points in the $N_{\max}$ generated points can fill the holes or missing parts by leveraging the generative prior in the position DDPM.

C Evaluation Details

Shading-FID. We follow [48] to calculate Shading-FID between rendered images of the generated meshes and meshes in the dataset. When generating meshes using our method, we do not use classifier-free guidance for the class condition. We use the released checkpoints of the baselines to generate meshes for evaluation. Since the baselines use different portions of the dataset for training[4], we use all data in a category including the training, validation, and test sets to render images as the reference set. We generate the same number of meshes as the reference set for evaluation. We follow the same procedures in [48] to render multi-view images of the generated meshes and meshes in the reference set, except that we render 512×512 images instead of 299×299 images to better capture details of the generated meshes and meshes in the reference set.

[48] uses Pyrender to render images by default, but we find that Pyrender-rendered images of the meshes generated by our method often bear obvious artifacts: There are many small dots on the mesh surfaces as shown in Figure 15. We validate that this is a problem caused by Pyrender by rendering the same meshes using MeshLab. We can see that images rendered by MeshLab do not bear any artifacts. Therefore, it is not because meshes generated by our method have any flaws. We hypothesize that this is because meshes generated by our method have too dense faces on their surfaces due to the face subdivision operation in the mesh autoencoder, and the vertex deformation operation in the refinement module could also result in extremely small faces. Pyrender can not render meshes with dense and small faces properly. We use Open3D to reduce the number of faces of the generated meshes to 50000, and use Pyrender to render the simplified mesh again. The result is shown in Figure 15. Indeed, we can

[4] They all use 70% or more data for training. Therefore, the comparison with our method, which uses 70% data for training, is fair.

Table 7: Architecture of the point-based encoders and upsampling modules in the mesh autoencoders.

	Input Points	Level 1	Level 2	Level 3	Level 4	Latent Points	Upsampled Points
Number of Points	16384	4096	1024	256	64	32-64	256
Feature Dimension	3	128	256	512	512	768	384
Number of Points	16384	4096	1024	256	-	128-256	1024
Feature Dimension	3	128	256	512	-	192	192
Number of Points	16384	4096	1024	-	-	512-1024	4096
Feature Dimension	3	128	256	-	-	48	128
Number of Points	16384	8192	4096	-	-	2048-4096	16384
Feature Dimension	3	128	256	-	-	12	3

Table 8: Reconstruction performance of the 4 mesh autoencoders. $N_{\min}$ and $N_{\max}$ are the minimum and maximum number of latent points in the latent space. D is the dimension of the features of the latent points.

$N_{\min}$	$N_{\max}$	D	CD $\times 10^{-3} \downarrow$	Mask-IoU $\uparrow$
32	64	768	1.55	0.954
128	256	192	1.34	0.968
512	1024	48	**1.07**	**0.971**
2048	4096	12	1.19	0.968

see that the artifacts are mitigated by reducing the number of faces. Therefore, we simplify our meshes to 50000 faces before calculating Shading-FID.

We also observe Pyrender-rendered images of meshes generated by baseline methods and find that they do not have this problem since their meshes typically have much fewer faces than our method, and they use either Marching Cube or DMTet to extract meshes, where extremely small faces are rare. The render result is shown in Figure 16. Therefore, the comparison between our method and baselines is fair.

1-NNA, MMD, Coverage. We follow [42] to compute 1-NNA, MMD, and Coverage between generated meshes and meshes in the dataset. For each category, we randomly sample 1000 meshes from the whole category as the reference set, and generate 1000 meshes for our method and baselines. We sample 2048 points from the mesh surfaces and normalize them to the bounding box $[-1, 1]^3$ to compute 1-NNA, MMD, and Coverage.

Meshes in the ShapeNet dataset are non-watertight and have inner structures in general. To sample points only from the mesh outer surface, we render multi-view depth maps for a mesh (100 random views), project each depth map to a point cloud using camera extrinsics and intrinsics, and concatenate them together to form a complete point cloud. To obtain a uniform point cloud, we first randomly downsample the concatenated point cloud to 16384 points and then downsample it to 2048 points using FPS. We use the same method to sample 2048 points from the generated meshes.

Table 9: The amount of computational resources (A100 GPUs) and time to train the position DDPMs and feature DDPMs. Note that the training time could be affected by many factors such as data I/O speed, GPU utilization, or node differences in a cluster.

N_{min}	N_{max}	Position DDPM		Feature DDPM	
		Number of GPUs	Days	Number of GPUs	Days
32	64	2	2	4	5
128	256	4	2	16	3
512	1024	8	3	16	4
2048	4096	32	8	32	12

Table 10: Ablation study of the number of latent points in the latent space. 1-NNA is reported.

N_{min}	N_{max}	1-NNA (CD) ↓				1-NNA (EMD) ↓			
		Airplane	Car	Chair	Table	Airplane	Car	Chair	Table
32	64	95.15	83.45	80.53	79.63	94.90	90.00	83.68	85.19
128	256	**70.70**	**85.35**	**53.00**	**53.90**	**68.80**	**82.50**	**52.85**	**54.35**
512	1024	76.30	86.60	74.72	75.53	74.85	84.60	76.03	75.23
2048	4096	99.90	99.75	99.30	99.05	99.90	99.70	99.34	98.59

D Ablation Study

We conduct an ablation study on the number of latent points in the latent space. We train 4 mesh autoencoders with different numbers of latent points $N_{\text{min}}, N_{\text{max}}$, and feature dimension D. We keep $N_{\text{max}} \times D$ fixed to ensure that the number of bits of the latent representation does not change. The architecture of the encoders and upsampling details are shown in Table 7. The architecture of decoders is the same as the one described in Section A.1. The mesh autoencoder with $N \in [128, 256]$ is trained for scratch as described in Section A.1. The other 3 autoencoders are initialized by using parameters from the same modules in the autoencoder with $N \in [128, 256]$, and parameters that do not exist in the autoencoder with $N \in [128, 256]$ are randomly initialized. The 3 autoencoders

Table 11: Ablation study of the number of latent points in the latent space. MMD is reported.

N_{min}	N_{max}	MMD (CD) ×1000 ↓				MMD (EMD) ×100 ↓			
		Airplane	Car	Chair	Table	Airplane	Car	Chair	Table
32	64	6.52	5.70	17.84	21.39	11.99	9.76	18.49	18.70
128	256	**3.50**	**3.99**	**14.8**	**18.53**	**8.31**	**7.93**	**15.73**	**15.45**
512	1024	3.75	4.46	17.83	23.53	8.75	8.26	17.83	17.99
2048	4096	44.47	35.83	150.23	162.67	30.31	23.69	53.29	54.85

Table 12: Ablation study of the number of latent points in the latent space. Coverage is reported.

$N_{\min}$	$N_{\max}$	COV (CD) (Percent) ↑				COV (EMD) (Percent) ↑			
		Airplane	Car	Chair	Table	Airplane	Car	Chair	Table
32	64	32.0	25.35	43.79	44.29	33.9	25.95	43.39	42.39
128	256	**47.6**	**28.6**	**49.45**	**50.65**	**49.0**	**31.1**	**48.95**	**48.55**
512	1024	39.8	24.7	29.33	30.43	41.4	26.0	28.43	27.83
2048	4096	2.4	1.4	2.8	3.4	2.4	1.8	2.6	3.0

Table 13: Average generation time (in seconds) per sample of our models tested on a single NVIDIA A100 GPU. The samples are generated in 1000 steps without any accelerations. Note that we have included the time to reconstruct meshes from latent points with features to feature DDPM's generation time.

$N_{\min}$	$N_{\max}$	Position DDPM	Feature DDPM	Total
32	64	0.12	0.22	0.34
128	256	0.54	0.63	1.17
512	1024	4.03	4.11	8.14
2048	4096	45.21	46.97	92.18

($N \in [32, 64], N \in [512, 1024], N \in [2048, 4096]$) are trained with learning rates $5 \times 10^{-4}, 2 \times 10^{-4}, 2 \times 10^{-4}$, respectively. They all use batchsize 128 and are trained until convergence: 400 epochs for $N \in [32, 64], N \in [512, 1024]$, and 500 epochs for $N \in [2048, 4096]$. The reconstruction performance of the 4 mesh autoencoders is shown in Table 8. We can see that all mesh autoencoders achieve relatively good reconstruction performance, and the mesh autoencoder with $N \in [512, 1024]$ achieves the best reconstruction performance.

Next, we train latent diffusion models in the latent space of these autoencoders. The DDPMs all use the same Transformer architecture described in Section A.2. The amount of computational resources and time to train the DDPMs are shown in Table 9. We use Shading-FID, 1NN-A, MMD, and Coverage to evaluate their generation performance. Shading-FID is shown in Table 5 in the main text, 1NN-A, MMD, and Coverage are shown in Table 10, Table 11, and Table 12, respectively. We can see that the diffusion model with $N \in [128, 256]$ achieves the best generation performance in most cases.

We also test the generation time of these latent diffusion models and the result is shown in Table 13. We can see that fewer latent points lead to faster generation. The model with $N \in [128, 256]$ achieves a good trade-off between generation quality and speed.

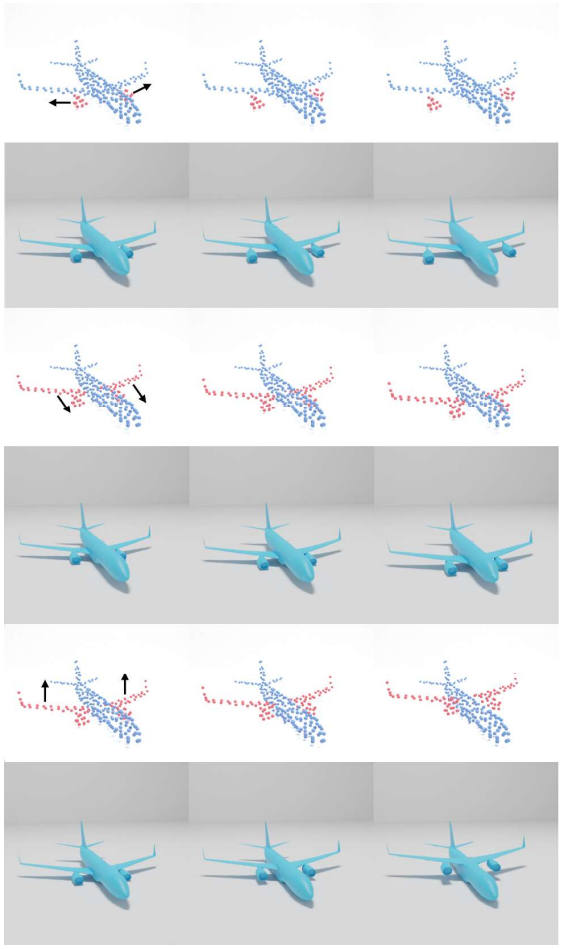

Fig. 17: Use positions of the latent points to control the generated shapes.

E Controllable Shape Generation

In Section 5.6 in the main text, we demonstrate the controllable generation ability of our method using the model with $N \in [512, 1024]$. In this section, we use our model with $N \in [128, 256]$ to demonstrate controllable generation. Several examples are shown in Figure 17. An example of shape interpolation is shown in Figure 18. Examples of shape combination are shown in Figure 19.

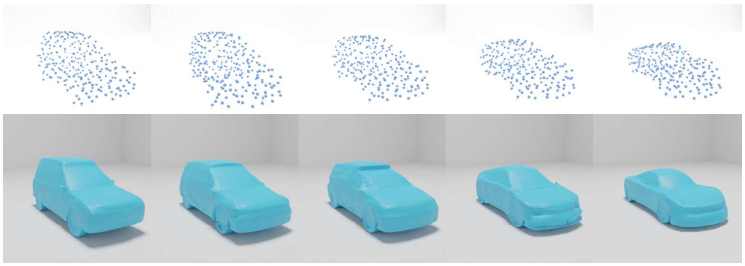

Fig. 18: An example of shape interpolation. The left-most is the source shape and the right-most is the target shape.

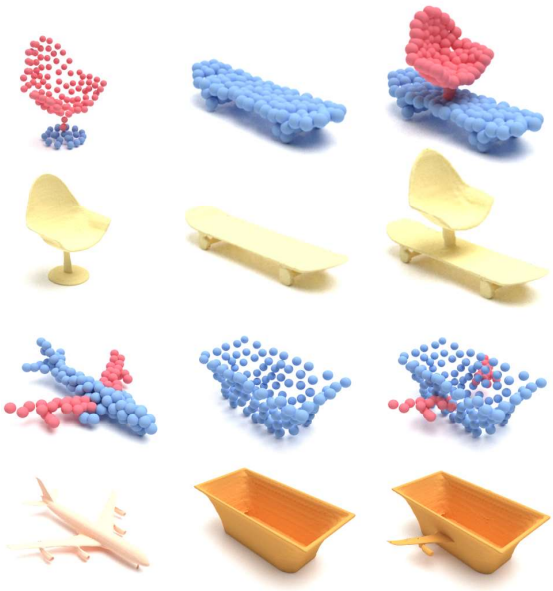

Fig. 19: Examples of shape combination.

F More Qualitative Results

We generate materials for meshes in Figure 1 using [41] and result is shown in Figure 20. We also qualitatively compare meshes generated by our method and baselines. Results are shown in Figure 21, Figure 22, Figure 23 Figure 24.

Fig. 20: Meshes generated by our method. Materials of the meshes are generated by an off-the-shelf material generator [41]. Note that texture or material generation is NOT the focus or contribution of this work. We merely intend to show that our method can be seamlessly combined with off-the-shelf methods to obtain meshes with textures or materials.

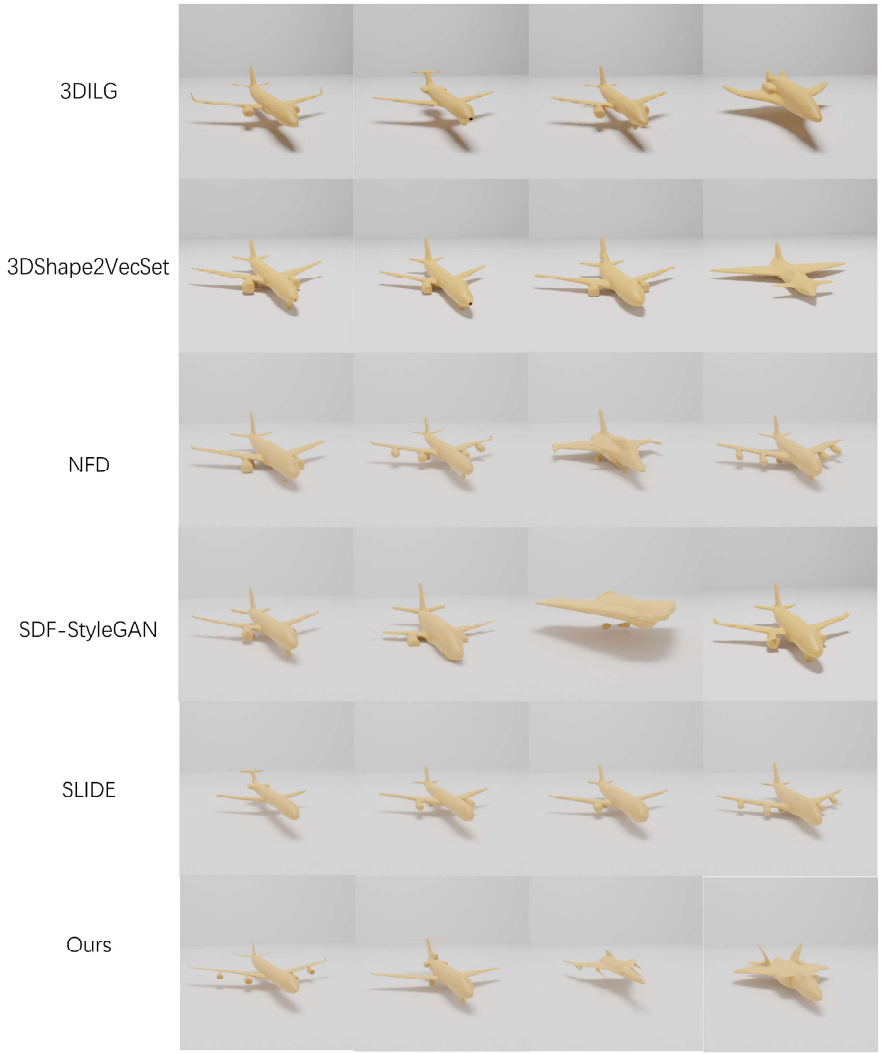

Fig. 21: Compare airplanes generated by our method and baselines. Notably, GetMesh better handles thin structures and fine details such as wings and engines than baselines.

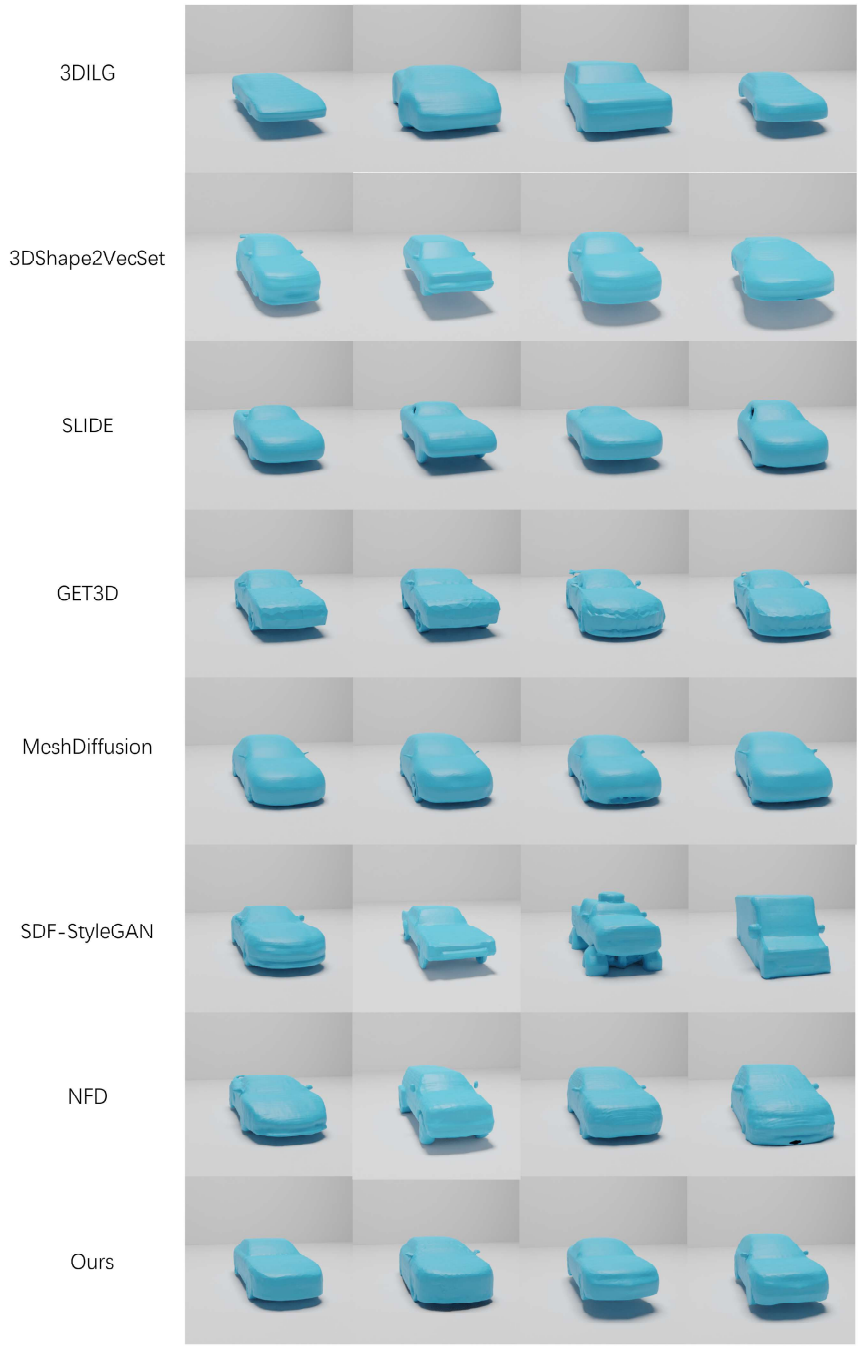

Fig. 22: Compare cars generated by our method and baselines. GetMesh generates meshes with smoother surfaces and sharper edges.

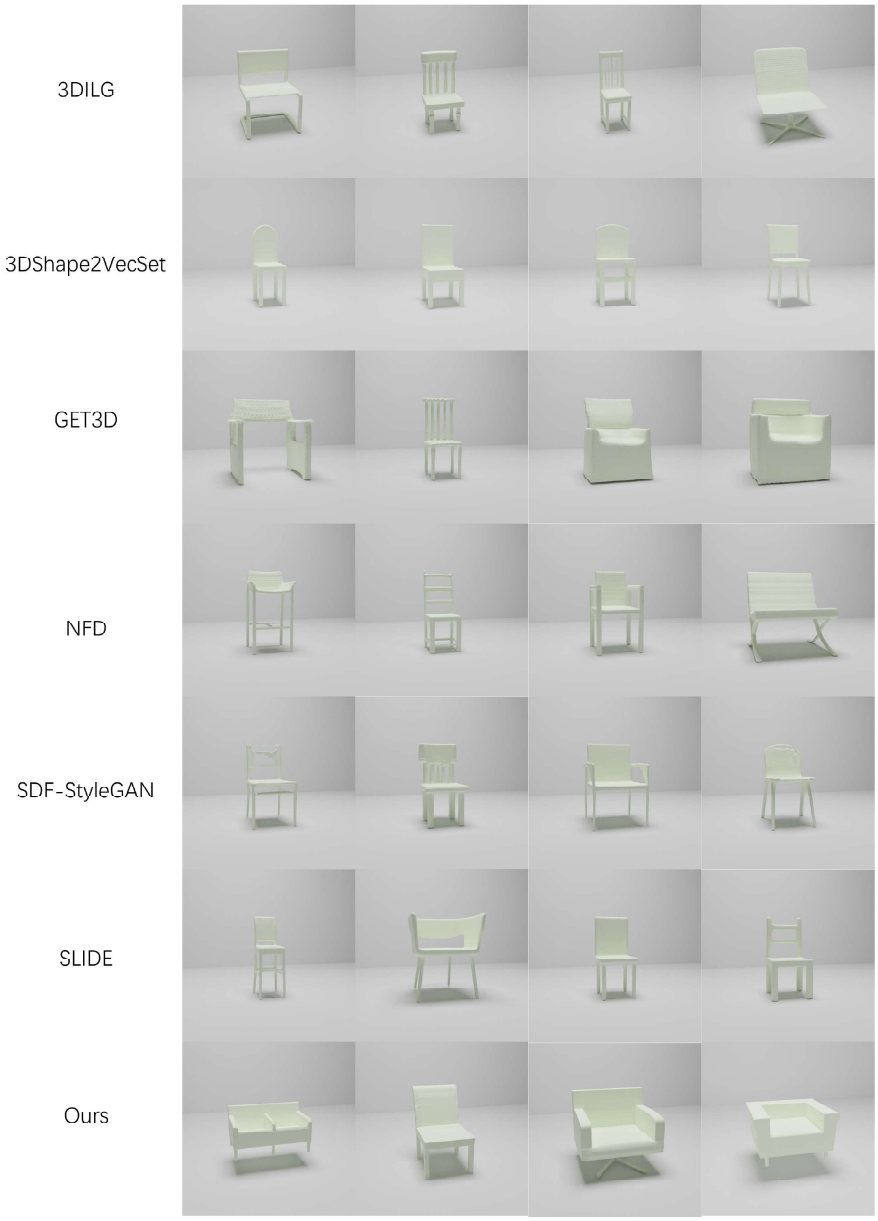

Fig. 23: Compare chairs generated by our method and baselines. GetMesh generates meshes with smoother surfaces and sharper edges.

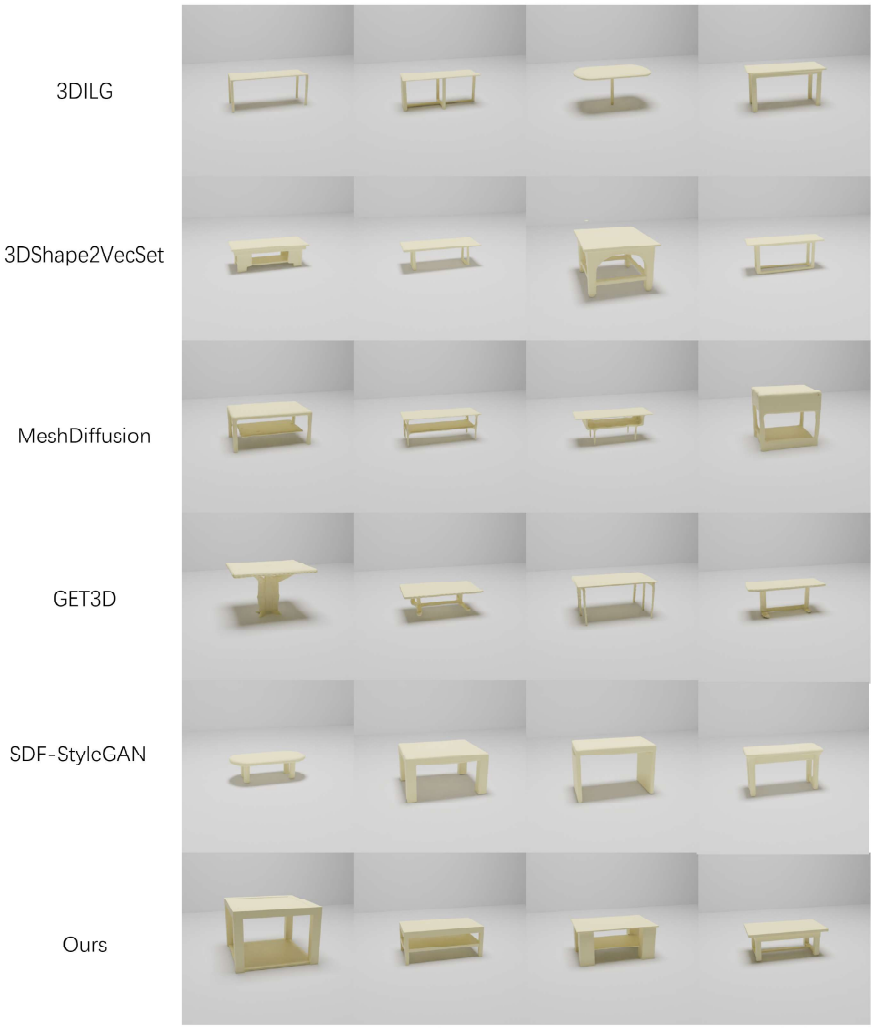

Fig. 24: Compare tables generated by our method and baselines. GetMesh generates meshes with smoother surfaces and sharper edges.