PointNSP: Autoregressive 3D Point Cloud Generation with Next-Scale Level-of-Detail Prediction

Supplementary Material

6. Permutation Invariance of Probability Distribution

Here we show why the distribution $p(\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_K) = \prod_{k=1}^{K} p(\mathbf{X}_k|\mathbf{X}_{k-1}, \ldots, \mathbf{X}_2, \mathbf{X}_1)$ in Eq. 3 preserves the permutation invariance property $p(\pi(\mathbf{x}_1, \ldots, \mathbf{x}_N)) = p(\mathbf{x}_1, \ldots, \mathbf{x}_N), \forall\, \pi \in S_N$ in Eq. 2.

By definition, the joint distribution factorizes as:

$$\begin{aligned} p(\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_K) = p(\mathbf{X}_1)p(\mathbf{X}_2|\mathbf{X}_1) \\ \ldots p(\mathbf{X}_K|\mathbf{X}_1, \ldots \mathbf{X}_{K-1}). \end{aligned} \tag{10}$$

Consider an arbitrary permutation π acting on the full set of points $\mathbf{X} = \bigcup_{k=1}^{K} \mathbf{X}_k$. This permutation can be decomposed into independent permutations within each scale:

$$\pi = (\pi_1, \pi_2, \ldots, \pi_K), \qquad \pi_k \in S_{s_k}. \tag{11}$$

Then we aim to prove that:

$$p(\pi_1(\mathbf{X}_1), \ldots, \pi_K(\mathbf{X}_K)) = p(\mathbf{X}_1, \ldots \mathbf{X}_K). \tag{12}$$

Recall that FPS is permutation-invariant, the resulting LoD sampling sequence $(\mathbf{X}_1, \ldots, \mathbf{X}_K)$ is *permutation-invariant* (its inherent stochasticity operates at the set level and is independent of point ordering). The core components of *PointNSP* —namely the feature encoder $\mathcal{E}(\cdot)$, upsampling, query, and the decoder $D(\cdot)$—are all *permutation-equivariant*, ensuring that each output feature remains aligned with its corresponding input point. Therefore, the mapping between $\mathbf{X}_k$ and conditioning context shapes $(\mathbf{X}_1, \ldots, \mathbf{X}_{k-1})$ remains unchanged with respect to any global permutation π. Then, for any permutation π_k of points within $\mathbf{X}_k$:

$$\begin{aligned} p(\pi_k(\mathbf{X}_k)|\pi_1(\mathbf{X}_1), \ldots \pi_{k-1}(\mathbf{X}_{k-1})) \\ = p(\mathbf{X}_k|\mathbf{X}_1, \ldots \mathbf{X}_{k-1}). \end{aligned} \tag{13}$$

This holds for each $k = 1, \ldots, K$. Then the joint distribution under these permutations is

$$\begin{aligned} & p(\pi_1(\mathbf{X}_1), \ldots, \pi_K(\mathbf{X}_K)) \\ &= \prod_{k=1}^{K} p(\pi_k(\mathbf{X}_k)|\pi_1(\mathbf{X}_1), \ldots \pi_{k-1}(\mathbf{X}_{k-1})) \\ &= \prod_{k=1}^{K} p(\mathbf{X}_k|\mathbf{X}_{k-1}, \ldots, \mathbf{X}_2, \mathbf{X}_1) \\ &= p(\mathbf{X}_1, \ldots, \mathbf{X}_K) \end{aligned} \tag{14}$$

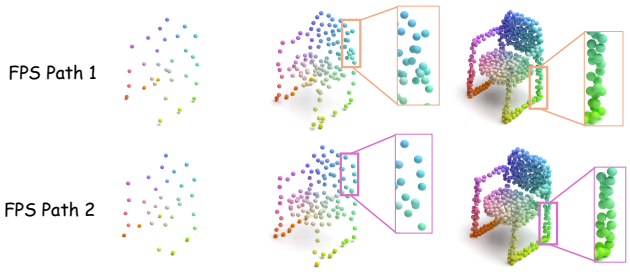

Figure S1. Illustration of different LoD sequences for a single shape. Owing to the inherent stochasticity of the sampling strategy in Eq. 17, our method naturally produces *diverse causal LoD sequences* for the same 3D shape across training epochs. At each scale, these variations encourage *broader spatial coverage* of the underlying geometry, thereby enhancing learning robustness.

Consequently, the autoregressive factorization preserves permutation invariance: permuting the points within any scale does not change the resulting joint probability:

$$p(\pi(\mathbf{x}_1, \ldots, \mathbf{x}_N)) = p(\mathbf{x}_1, \ldots, \mathbf{x}_N), \forall\, \pi \in S_N. \tag{15}$$

7. Algorithm Details

LoD Sequence Sampling. This step is a core component of *PointNSP* training, and we provide additional details here to eliminate any potential ambiguity. If one were to directly apply the original image-based VAR framework [54] to 3D point clouds, latent features for sampled LoD sequences would be obtained via

$$\mathbf{f}_k = \mathrm{FPS}(\mathbf{f}^{k-2} - \tilde{\mathbf{f}}_{k-1}), \mathbf{f}_1 = \mathrm{FPS}(\mathbf{f}^0). \tag{16}$$

However, this strategy is unsuitable for 3D point clouds. Applying FPS directly in latent space implicitly assumes that latent distances form a meaningful metric, yet these distances need not correlate with underlying geometric structure. As a result, sampling in latent space cannot reliably preserve geometric uniformity or spatial coverage. To address this, rather than iteratively applying FPS in latent space, we apply FPS in Euclidean 3D space in a *fine-to-coarse* manner:

$$\mathbf{X}_{k-1} = \mathrm{FPS}(\mathbf{X}_k), \mathbf{X}_K = \mathbf{X}. \tag{17}$$

This produces a geometrically consistent LoD sequence $(\mathbf{X}_1, \ldots, \mathbf{X}_K)$ from coarse to fine, with guaranteed point correspondences $\mathbf{X}_1 \subsetneq \cdots \subsetneq \mathbf{X}_{K-1} \subsetneq \mathbf{X}_K$. We then obtain latent features by querying the indices mapping each

$\mathbf{X}_k$ back to the original point set:

$$\mathbf{f}_k = \text{query}(\mathbf{f}^{k-2} - \tilde{\mathbf{f}}_{k-1}, \mathbf{X}_k),\ \mathbf{f}_1 = \text{query}(\mathbf{f}^0, \mathbf{X}_1), \quad (18)$$

where $\mathbf{f}^0 = \mathcal{E}(\mathbf{X})$. This is Eq. 4. Here we need to emphasize that the LoD sequence $(\mathbf{X}_1, \ldots, \mathbf{X}_K)$ is constructed in advance following Eq. 17. Since FPS is inherently stochastic due to its random initialization, we can naturally obtain diverse LoD sequences for each shape $\mathbf{X}$ across training epochs. This variability is desirable, as the refinement path need not be unique; sampling multiple paths improves spatial coverage, as discussed in Section 3.2. For example, for epoch 0 and epoch 1, we may obtain two LoD sequences for shape $\mathbf{X}$:

$$(\mathbf{X}_1^0, \ldots, \mathbf{X}_K^0),\ (\mathbf{X}_1^1, \ldots, \mathbf{X}_K^1). \quad (19)$$

At each scale, $\mathbf{X}_k^0$ and $\mathbf{X}_k^1$ represent the same underlying surface but with different point coverages, which helps improve learning robustness and generalization. We illustrate this LoD sequence sampling issue in Figure S1.

Coordinate-based Positional Encoding. We adopt the absolute positional encoding strategy purely based on 3D coordinates used in TIGER [47]. Based on our experiments, we find the Base λ Position Encoding (BλPE) performs better and here we present its formula:

$$p = \lambda^2 * z_i + \lambda * y_i + x_i \quad (20)$$

$$\mathbf{P}_k(p, 2i) = \sin(\frac{p}{10000^{\frac{2i}{D}}}) \quad (21)$$

$$\mathbf{P}_k(p, 2i+1) = \cos(\frac{p}{10000^{\frac{2i}{D}}}), \quad (22)$$

where $\mathbf{x}_i = \mathbf{X}_k[i] = (x_i, y_i, z_i) \in \mathbb{R}^3$, p is a polynomial expression with hyperparameter coefficient λ. We set $\lambda = 1000$ following the setting in TIGER [47], which means this preserves three decimal places of precision. Here $\mathbf{P}_k \in \mathbb{R}^{s_k \times d}$ denotes the positional embedding of all tokens within the scale k. In short, we apply the BλPE embedding strategy scale-by-scale.

Intra-Scale Token Embedding. Instead of simply adding all token embeddings together, the real implementation is analogous to Llama [10] by adding positional embedding and scale embedding to query and key vectors. Specifically, we retrieve the codebook embedding $\mathbf{z}_k$ for each scale token q_k and then upsampled from input scale to output scale $(s_k \rightarrow s_{k+1})$: $\mathbf{z}_k = \text{upsampling}(\mathbf{z}_k) \in \mathbb{R}^{s_{k+1} \times d}$ following the Eq. 6. Positional embedding $\mathbf{p}_k^i = \mathbf{P}_k[i]$ for each token q_k^i is derived with from the decoded intermediate structure $\mathbf{X}_k$ as described in Eq. 9. During inference stage, the intermediate structure $\hat{\mathbf{X}}_k$ is decoded using the predicted token $\hat{q}_k$ instead. Additionally, the model needs to know which scale that each token belongs to. Therefore, $\mathbf{s}_k$ is a simple one-hot embedding $\mathbf{s}_k = \text{one-hot-embedding}(k)$ out of total K scales. Tokens from the same scale k share the same scale embedding $\mathbf{s}_k$ ($\mathbf{s}_k^i = \mathbf{s}_k^j$ for q_k^i, q_k^j). For all input scale tokens, we add both the positional embedding $\mathbf{p}_k^i$ and the scale embedding $\mathbf{s}_k^i$ to the query $\mathbf{u}_k^i$ and key vectors $\mathbf{v}_k^i$ derived in the attention mechanism:

$$\mathbf{u}_k^i = \mathbf{W_U}\mathbf{z}_k^i + \mathbf{p}_k^i + \mathbf{s}_k^i, \mathbf{v}_k^i = \mathbf{W_V}\mathbf{z}_k^i + \mathbf{p}_k^i + \mathbf{s}_k^i, \quad (23)$$

where $\mathbf{W_U}$ and $\mathbf{W_V}$ are projection matrices for queries and keys respectively. Together with value vectors, these vectors will be fed to the block-wise causal transformer for next-scale token prediction.

8. Hyperparameters & Reproducibility Settings

Hyperparameters & Reproducibility Settings. Specifically, we set the learning rate $3e^{-4}$ and the batch-size 32. We perform all the experiments on a workstation with Intel Xeon Gold 6154 CPU (3.00GHz) and 8 NVIDIA Tesla V100 (32GB) GPUs. We use an AdamW optimizer with an initial learning rate of 10^{-4} for VQVAE training and 10^{-3} for autoregressive transformer training respectively. For upsampling and completion experiments, we follow the experimental settings of PVD [69]. For many-class generation, we mainly inherit the experimental setting from LION [65].

Hyperparameter	**Value**
# PVCNN layers	4
# PVCNN hidden dimension	1024
# PVCNN voxel grid size	32
# MLP layers	6
# Attention dimension	1024
# Attention head	32
Optimizer	AdamW
Weight Decay	0.01
LR Schedule	Cosine

The effect of # scales K. Figure S2 illustrates the impact of the total number of scales on the overall performance of *PointNSP*. As the number of scales increases, *PointNSP*'s performance improves accordingly. However, beyond $K = 11$ scales, no further performance gains are observed. We hypothesize that additional scales may require higher point cloud densities to be effective. Moreover, increasing the number of scales naturally leads to longer sampling times.

9. More Experimental Results

Due to page limitations, the main paper reports only the strongest baseline methods in the primary comparison ta-

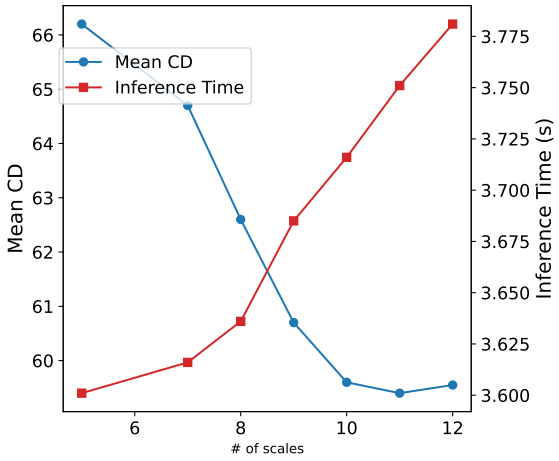

Figure S2. The effect of number of scales on the overall performance of *PointNSP*.

ble. Here, we provide a more comprehensive evaluation that additionally includes 1-GAN [1] (GAN-based), PointFlow [62], DPF-Net [21], SoftFlow [19] (normalizing flow–based), SetVAE [20] (VAE-based), PVD-DDIM [50] (diffusion models with advanced samplers), PSF [58] (flow-matching–based), and DIT-3D [36]. We include only methods with publicly available implementations or those reporting explicit quantitative results in their original papers. Please see Table S3 for comprehensive comparisons.

Since ShapeNet categories are highly imbalanced, prior works typically report results only on the three largest categories—airplane, chair, and car—while the remaining categories contain significantly fewer samples. To provide a more comprehensive evaluation, we additionally report performance on three smaller categories: table, sofa, and lamp. As shown in Table S1, *PointNSP* consistently and substantially outperforms all baseline methods, demonstrating its strong learning efficiency *under limited data conditions*.

Model	Table	Sofa	Lamp
	CD / EMD	CD / EMD	CD / EMD
PointFlow [62]	92.3 / 92.8	85.6 / 88.4	88.7 / 98.5
PointGPT [4]	91.5 / 90.9	83.8 / 88.4	88.7 / 98.5
ShapeGF [2]	83.7 / 81.5	79.2 / 81.3	86.4 / 86.8
PVD [69]	76.8 / 80.2	73.5 / 96.7	89.1 / 89.4
LION [65]	68.4 / 78.7	67.9 / 90.2	81.3 / 82.6
***PointNSP*-m**	**61.2 / 76.5**	**60.8 / 83.7**	**73.9 / 74.8**

Table S1. Per-category generation quality on ShapeNet. We report 1-NNA CD/EMD (↓) on **Table**, **Sofa**, and **Lamp**. Best in **bold**, second best underlined.

In addition to the 55-class generation setting, we also evaluate the 13-class setting used in LION [65]. The results, presented in Table S2, show that *PointNSP* achieves state-of-the-art generation quality.

Model	CD ↓	EMD ↓
TreeGAN [49]	96.80	96.60
PointFlow [62]	63.25	66.05
ShapeGF [2]	55.65	59.00
SetVAE [20]	79.25	95.25
PDGN [17]	71.05	86.00
DPF-Net [21]	67.10	64.75
DPM [34]	62.30	86.50
PVD [69]	58.65	57.85
LION [65]	55.52	53.89
***PointNSP*-m**	**54.70**	**52.82**

Table S2. Generation results (1-NNA ↓) trained jointly on 13 classes of ShapeNet-vol.

10. More Visualization Results

We showcase diverse 3D point clouds generated by *PointNSP* across a wide variety of shapes (Figures S3 and S4). Additional single-class generation results for five categories are provided in Figures S5–S9. We further illustrate the multi-scale sequential generation process in Figures S10–S11, and present more examples of point cloud completion and upsampling in Figures S13 and S12.

11. Other Related Works

Point Cloud Upsampling. Point cloud upsampling is a crucial process in 3D modeling, aimed at increasing the resolution of low-resolution 3D point clouds. PU-Net [63] pioneered the use of deep neural networks for this task, laying the foundation for subsequent advancements. Models such as PU-GCN [41] and PU-Transformer [45] have further refined point cloud feature extraction by leveraging graph convolutional networks and transformer networks, respectively. Additionally, approaches like Dis-PU [25], PU-EVA [33], and MPU [9] have enhanced the PU-Net pipeline by incorporating cascading refinement architectures. Other methods, such as PUGeo-Net [43], NePs [13], and MAFU [44], employ local geometry projections into 2D space to model the underlying 3D surface. More recent approaches have reframed upsampling as a generation task. For instance, PU-GAN [24] and PUFA-GAN [27] leverage generative adversarial networks (GANs) to produce high-resolution point clouds. Grad-PU [15] first generates coarse dense point clouds through nearest-point interpolation and then refines them iteratively using diffusion models. In contrast, PUDM [46] directly utilizes conditional diffusion models, treating sparse point clouds as input conditions for generating denser outputs. In this work, our

Table S3. The *Performance* (1-NNA) is evaluated based on single-class generation. The second block specifies the types of generative models used in each study. The best performance is highlighted in **bold**, while the second-best performance is underlined. Performance is reported on two dataset splits: the top corresponds to the random split, and the bottom corresponds to the LION split.

Model	Generative Model	Airplane		Chair		Car		Mean CD ↓	Mean EMD ↓
		CD ↓	EMD ↓	CD ↓	EMD ↓	CD ↓	EMD ↓		
1-GAN [1]	GAN	87.30	93.95	68.58	83.84	66.49	88.78	74.12	88.86
PointFlow [62]	Normalizing Flow	75.68	70.74	62.84	60.57	58.10	56.25	65.54	62.52
DPF-Net [21]	Normalizing Flow	75.18	65.55	62.00	58.53	62.35	54.48	66.51	59.52
SoftFlow [19]	Normalizing Flow	76.05	65.80	59.21	60.05	64.77	60.09	66.67	61.98
SetVAE [20]	VAE	75.31	77.65	58.76	61.48	59.66	61.48	64.58	66.87
ShapeGF [2]	Diffusion	80.00	76.17	68.96	65.48	63.20	56.53	70.72	66.06
DPM [34]	Diffusion	76.42	86.91	60.05	74.77	68.89	79.97	68.45	80.55
PVD-DDIM [50]	Diffusion	76.21	69.84	61.54	57.73	60.95	59.35	66.23	62.31
PSF [58]	Diffusion	74.45	67.54	58.92	54.45	57.19	56.07	62.41	57.20
PVD [69]	Diffusion	73.82	64.81	56.26	53.32	54.55	53.83	61.54	57.32
LION [65]	Diffusion	72.99	64.21	55.67	53.82	53.47	53.21	61.75	57.59
DIT-3D [36]	Diffusion	-	-	54.58	53.21	-	-	-	-
TIGER [47]	Diffusion	73.02	64.10	55.15	53.18	53.21	53.95	60.46	57.08
PointGrow [52]	Autoregressive	82.20	78.54	63.14	61.87	67.56	65.89	70.96	68.77
CanonicalVAE [7]	Autoregressive	80.15	76.27	62.78	61.05	63.23	61.56	68.72	66.29
PointGPT [4]	Autoregressive	74.85	65.61	57.24	55.01	55.91	54.24	63.44	62.24
PointNSP-s (*ours*)	Autoregressive	72.92	63.98	54.89	53.02	52.86	52.07	60.22	56.36
***PointNSP*-m (*ours*)**	**Autoregressive**	**72.24**	**63.69**	**54.54**	**52.85**	**52.17**	**51.85**	**59.65**	**56.13**
LION	Diffusion	67.41	61.23	53.70	52.34	53.41	51.14	58.17	54.90
TIGER	Diffusion	67.21	56.26	54.32	51.71	54.12	50.24	58.55	52.74
PointNSP-s (*ours*)	Autoregressive	67.15	56.12	54.22	51.19	53.98	50.15	58.45	52.49
***PointNSP*-m (*ours*)**	**Autoregressive**	**66.98**	**56.05**	**54.01**	**53.76**	**53.12**	**50.09**	**58.04**	**52.30**

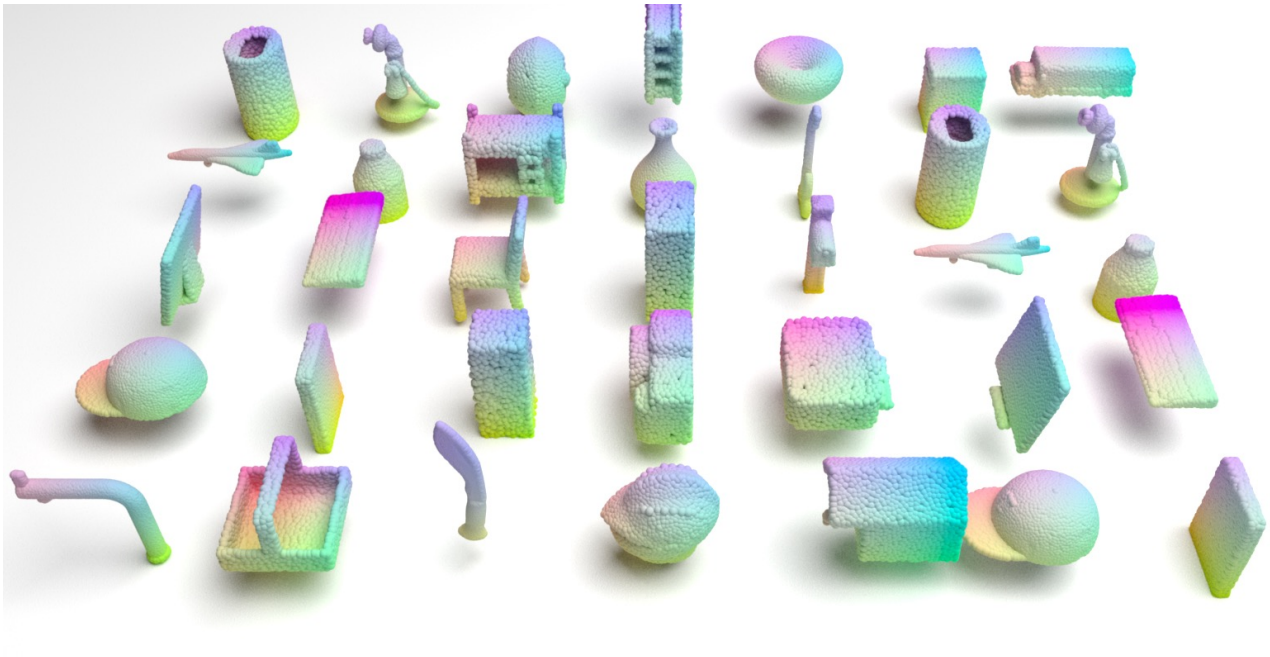

Figure S3. Generated shapes from the *PointNSP* model trained on ShapeNet's other categories.

generative model, *PointNSP*, incorporates upsampling networks in both two training stages, making it well-suited for enhancing downstream point cloud upsampling tasks.

12. Limitations & Broader Impact

PointNSP does not exhibit major limitations, though a primary challenge lies in learning high-quality multi-scale codebook embeddings for 3D point cloud representations, particularly in avoiding codebook collapse. Several promis-

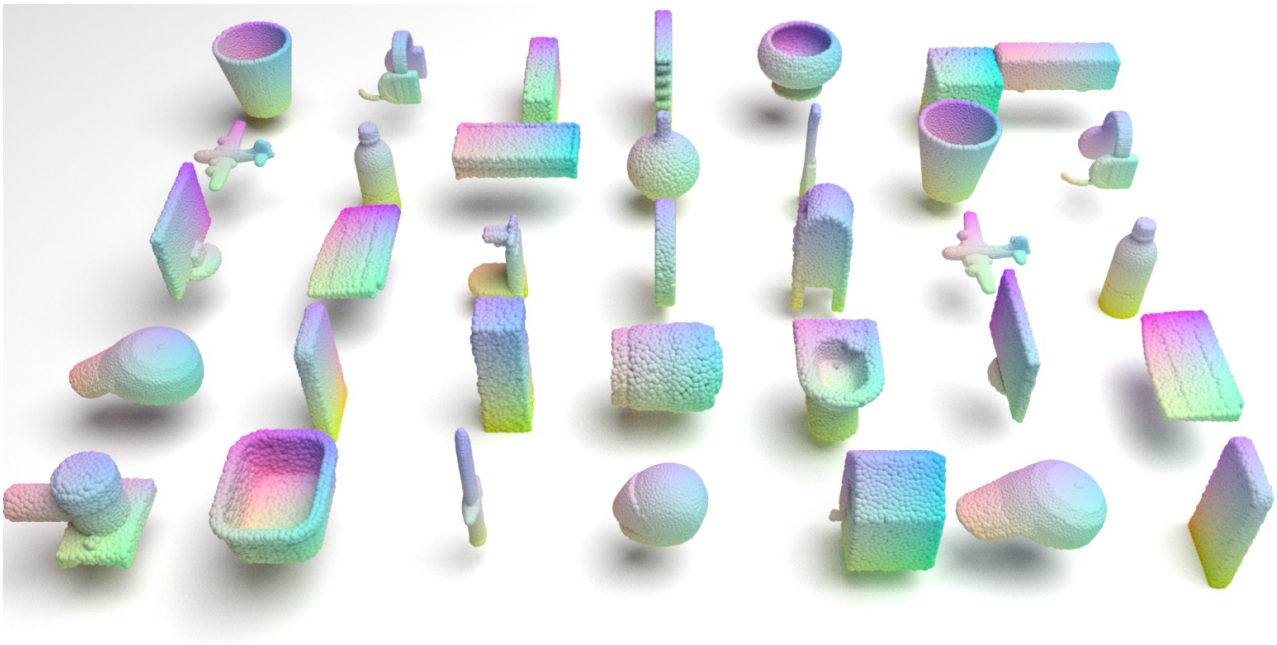

Figure S4. Generated shapes from the *PointNSP* model trained on ShapeNet's other categories.

ing directions arise for future work. One avenue is scaling generation toward ultra-dense point clouds (e.g., 10k–100k points), which could subsequently be converted into high-fidelity meshes. Another is enabling fine-grained control over local geometric structures, a capability crucial for practical deployment. Although this work does not present immediate societal risks, potential misuse for generating harmful 3D content warrants careful consideration by the broader community. From a research standpoint, *PointNSP* represents a significant contribution to both the generative modeling and 3D point cloud communities.

Figure S5. Generated single-class shapes from the *PointNSP* model trained on ShapeNet's other categories.

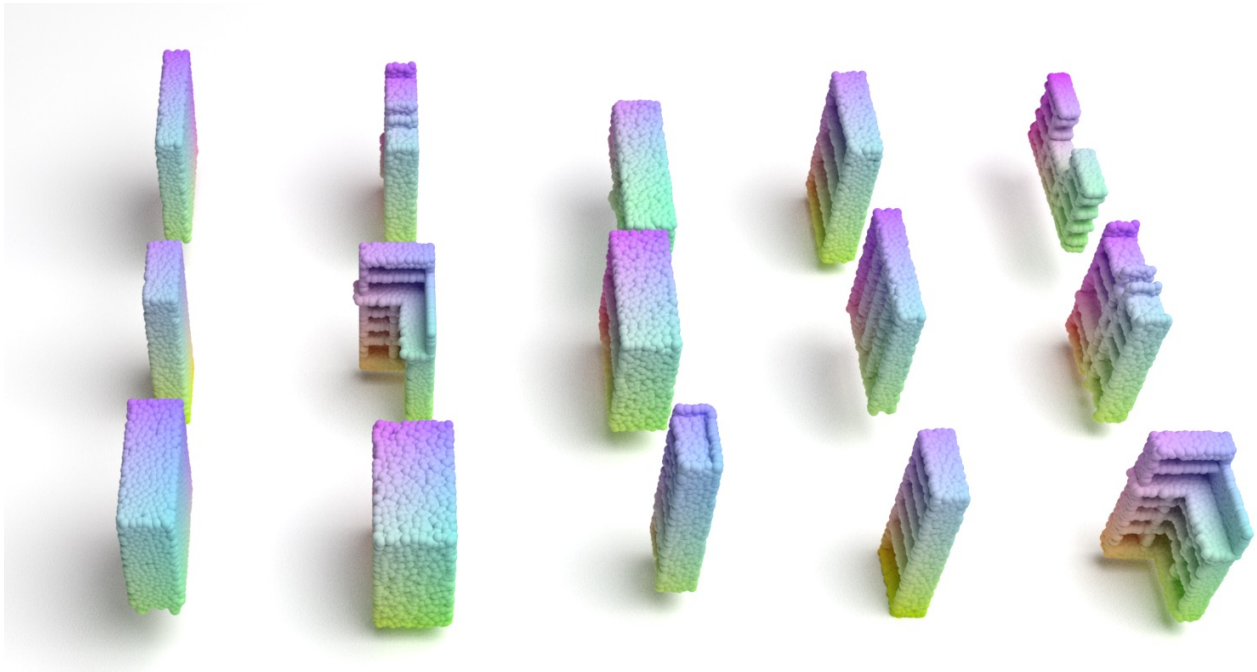

Figure S6. Generated single-class shapes from the *PointNSP* model trained on ShapeNet's other categories.

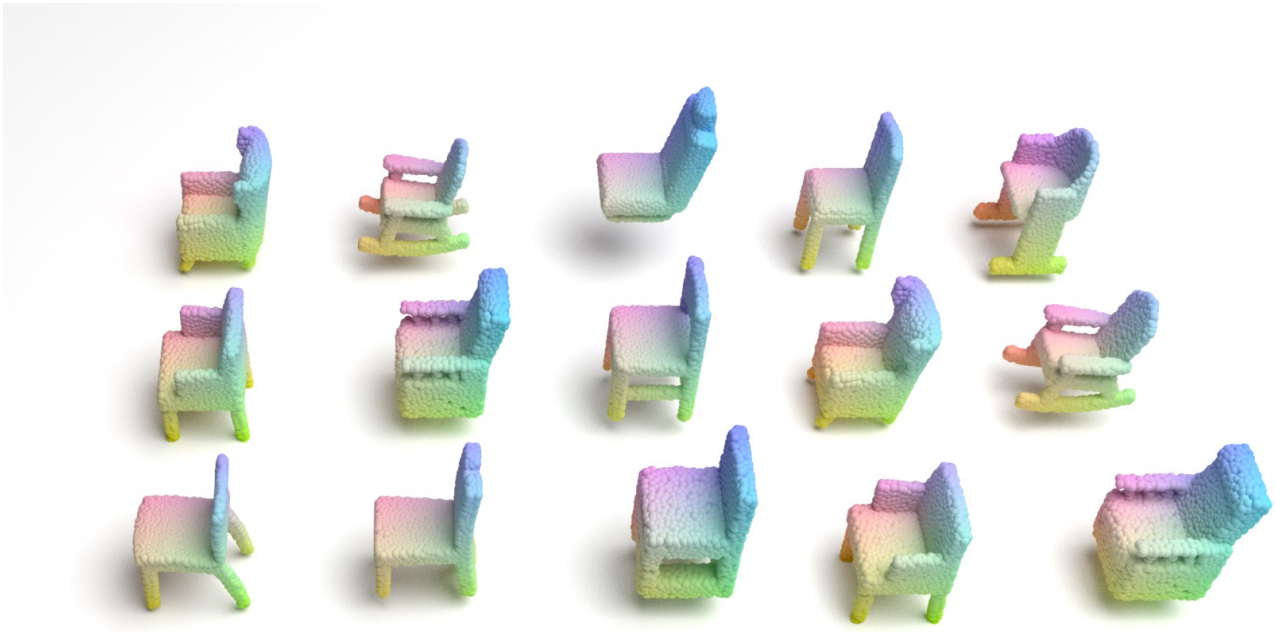

Figure S7. Generated single-class shapes from the *PointNSP* model trained on ShapeNet's other categories.

Figure S8. Generated single-class shapes from the *PointNSP* model trained on ShapeNet's other categories.

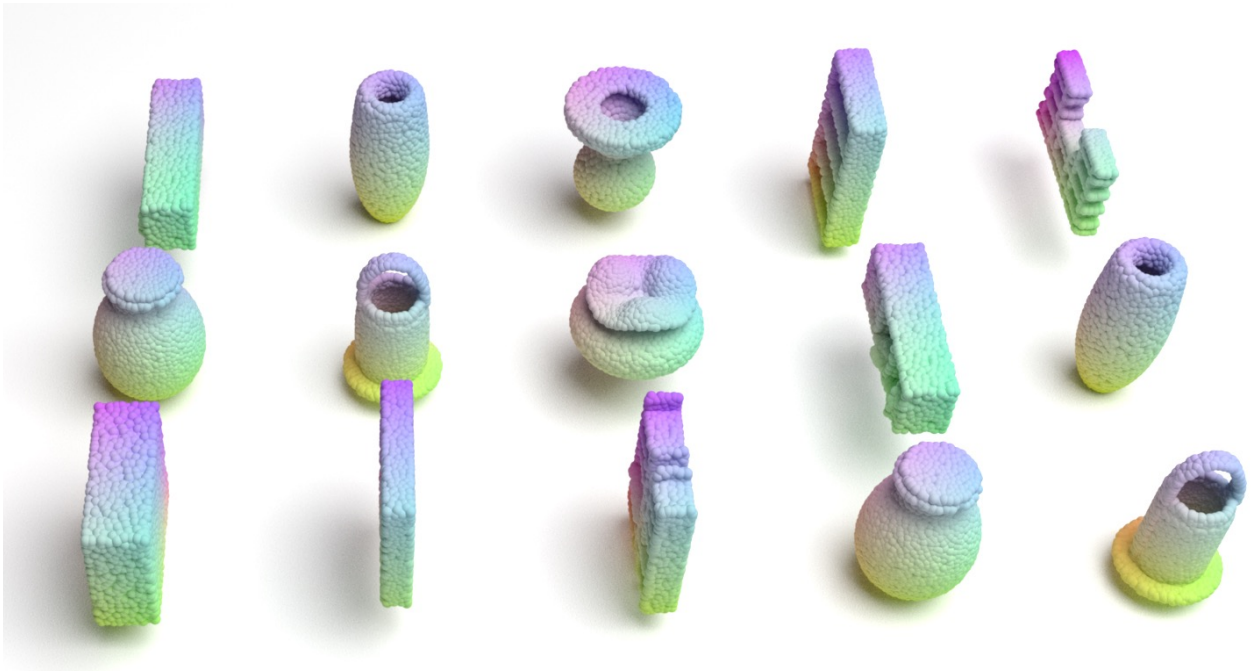

Figure S9. Generated single-class shapes from the *PointNSP* model trained on ShapeNet's other categories.

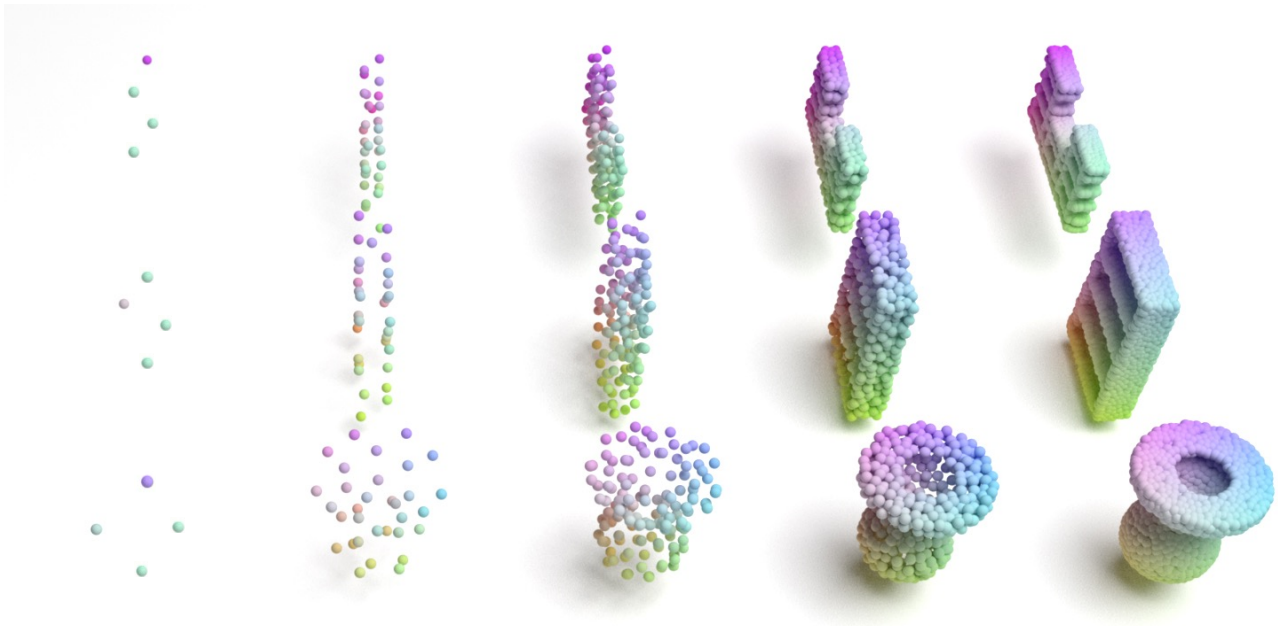

Figure S10. Illustration of multi-scale point cloud generation from the *PointNSP* model.

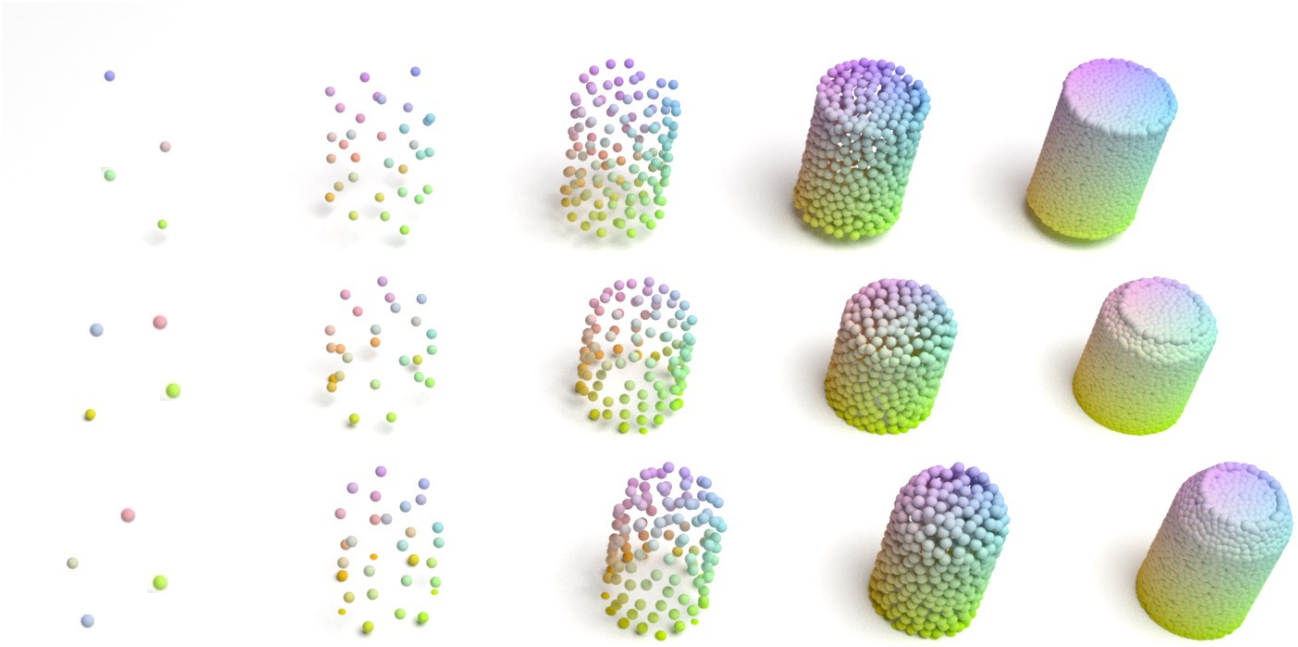

Figure S11. Illustration of multi-scale point cloud generation from the *PointNSP* model.

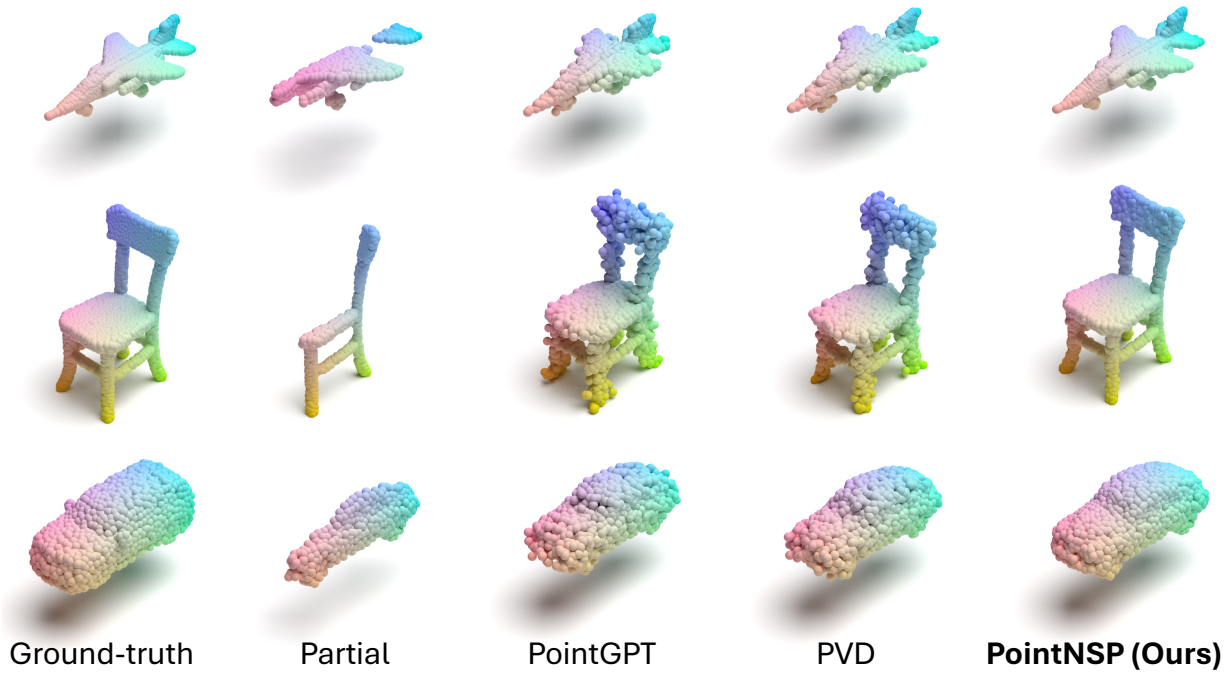

Figure S12. Visualizations of more point cloud completion results.

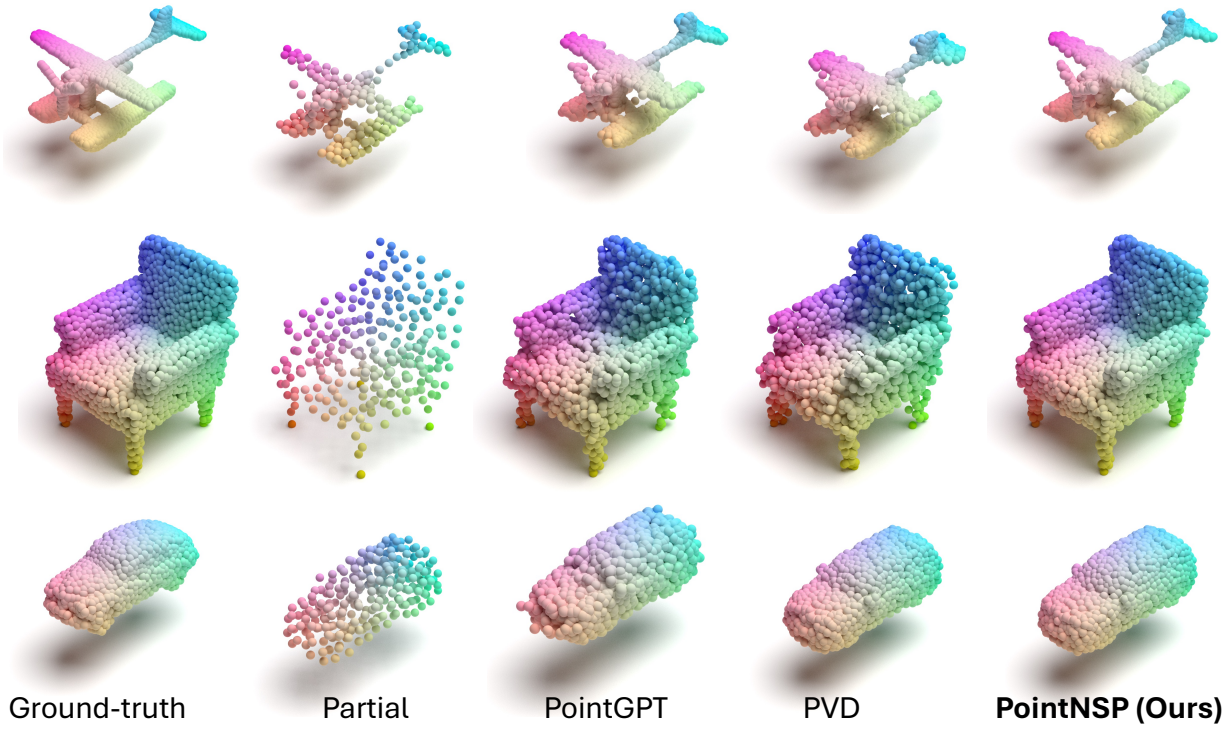

Figure S13. Visualizations of more point cloud upsampling results.