Geometry Distributions

Supplementary Material

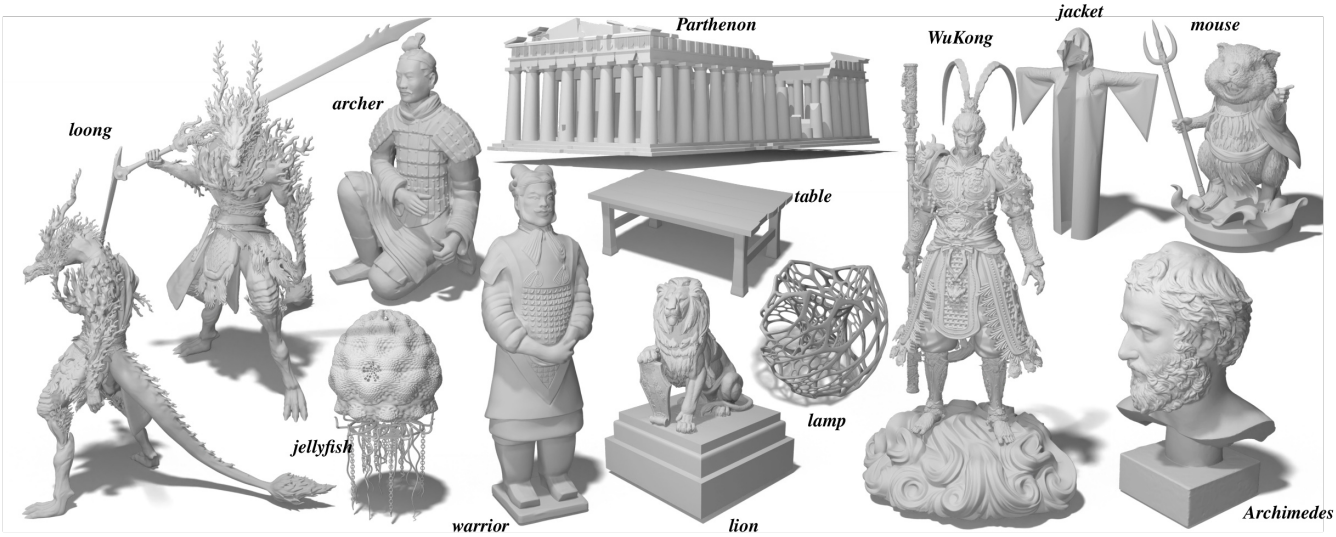

Figure 16. The ground-truth geometries for the shapes shown in Fig. 1.

A. Additional results

Fig. 17 extends Fig. 3, demonstrating that our approach generates more uniformly distributed samples with higher fidelity across different resolutions, compared to the vector field-based method. Fig. 19 (an extension of Tab. 2c), and Fig. 18 (an extension of Fig. 14), provide error visualization of L_2 distance to surface, demonstrating how the sampling steps affect the forward sampling and inverse sampling, respectively, for geometry recovery. In Fig. 20 and Fig. 21, we show additional results of using our method to represent textured geometry and high-resolution scene. In Fig. 26 we show additional results to justify the inversion process discussed in Sec. 3.2 and Sec. 4.4: we composite inverse sampling and forward sampling from *different* surfaces, yet still obtain expected results. Specifically, the composed sampling process allows us to transform the WuKong shape into a jellyfish, lamp, sphere, or a plane. In Tab. A.1 we report the mesh complexity of the experimented shapes.

name	*mesh stats*		
	# vtx (*K*)	*# face* (*K*)	*disk storage* (*Mb*)
WuKong	19226	6408	878
Archimedes	6162	2054	276
loong	3956	1318	176
jellyfish	3908	1302	178
mouse	793	1423	58
archer	710	1420	58
lamp	478	159	21
warrior	433	866	35
lion	303	606	24
jacket	146	49	11
city	249	83	11
Parthenon	117	39	5
valley	74	37	6
Spot	3.2	5.8	0.37
table	0.82	0.73	0.07

Table A.1. We report the mesh complexity and storage cost for the experimented shapes. Unless explicitly mentioned in the table, the shapes can be found in Fig. 16.

B. Implementation details

B.1. Mesh normalization

We normalize all the meshes using the following pseudo-code. First, we sample 10 million points on the surface. Next, we shift and scale the mesh based on the mean and standard deviation of the sampled points. As a result, the surface points are approximately centered around zero with unit variance. We found that this normalization is effective in stabilizing the training process.

```
points, _ = trimesh.sample.sample_surface(mesh,
    10000000)
mesh.vertices -= points.mean()
mesh.vertices /= points.std()
```

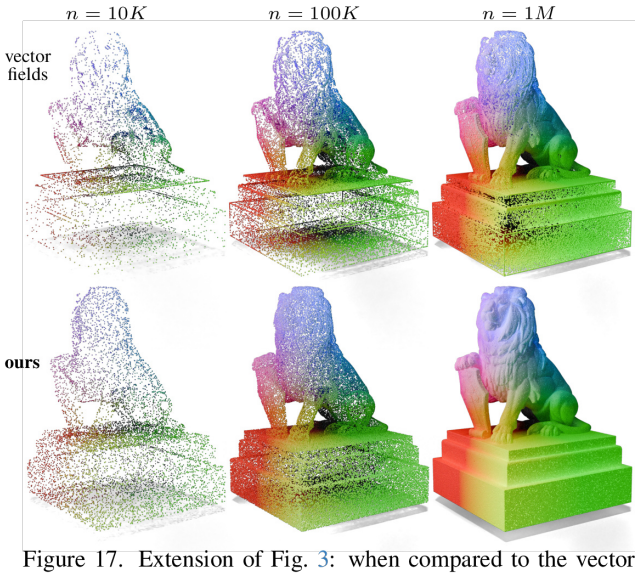

Figure 17. Extension of Fig. 3: when compared to the vector fields-based method (*top*), our **GEOMDIST** (*bottom*) produces more uniformly distributed samples with higher fidelity.

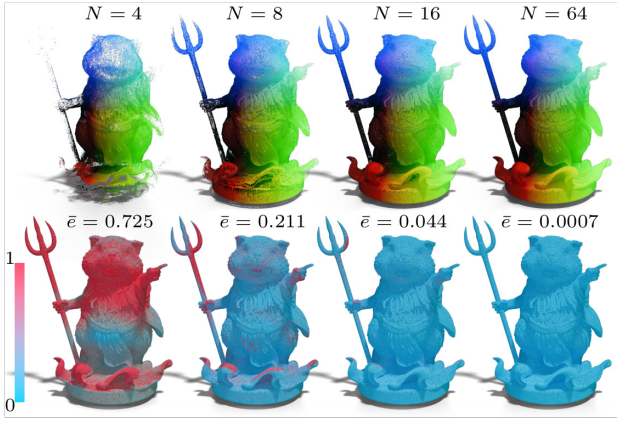

Figure 18. *Top*: the recovered shape from inversion $\mathcal{E} \circ \mathcal{D}(\mathbf{x})$ using different number of inverse sampling steps as in Eq. (4). *Bottom*: per-sample inversion error (colored on the original shape) and the average error $\bar{e}$ (L_2 distance to the surface).

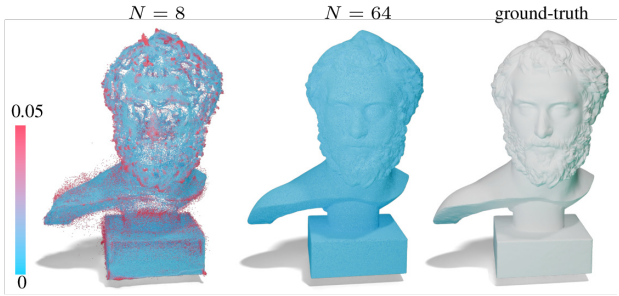

Figure 19. Errors (L_2 distance to the surface) of 1 million points using different sampling steps N (see Eq. (3)). *Left*: $N = 8$. *Middle*: $N = 64$. *Right*: ground-truth mesh. Per-point errors (colored by L_2 distance to the ground-truth surface) are visualized. Results for $N = 16, 32$ are omitted, as they are nearly identical (see Tab. 2c).

Figure 20. Applications on textured geometries. The setup is the same with Fig. 10. We show 1 million sampled points. *Top:* ground-truth. *Bottom:* ours.

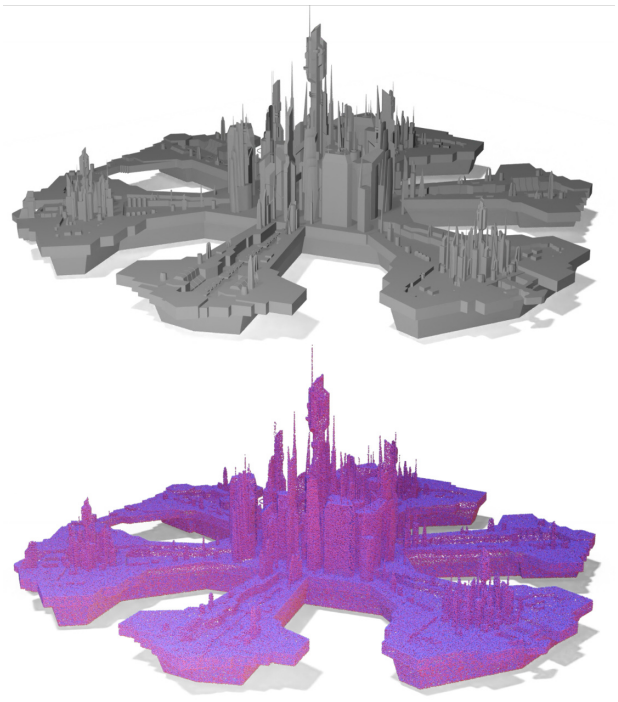

Figure 21. Results on city. *Top:* ground-truth. *Bottom:* ours.

B.2. Uniform distribution

When using uniform distribution as the initial noise in diffusion models (e.g., see Fig. 5), we scale the samples from uniform distribution to have zero mean and unit variance.

```
n = (torch.rand_like(x) - 0.5) / np.sqrt(1/12)
```

B.3. Chamfer distance

We use Chamfer distance to measure the distance between samples from our Geometry distribution, $\mathcal{X}_{\text{gen}}$, and the samples from ground-truth surface, $\mathcal{X}_{\text{ref}}$, to quantify the accuracy. This is defined as:

$$\text{ChamferDist}(\mathcal{X}_{\text{ref}}, \mathcal{X}_{\text{gen}}) = \frac{1}{|\mathcal{X}_{\text{ref}}|} \sum_{\mathbf{a} \in \mathcal{X}_{\text{ref}}} \min_{\mathbf{b} \in \mathcal{X}_{\text{gen}}} \|\mathbf{a} - \mathbf{b}\|_2 + \frac{1}{|\mathcal{X}_{\text{gen}}|} \sum_{\mathbf{b} \in \mathcal{X}_{\text{gen}}} \min_{\mathbf{a} \in \mathcal{X}_{\text{ref}}} \|\mathbf{a} - \mathbf{b}\|_2 .$$

The python code for calculating the Chamfer distance is as follows:

```
# prediction: B x 3
# reference: B x 3
from scipy.spatial import cKDTree as KDTree
tree = KDTree(prediction)
dist, _ = tree.query(reference)
d1 = dist
gt_to_gen_chamfer = np.mean(dist)
gt_to_gen_chamfer_sq = np.mean(np.square(dist))

tree = KDTree(reference)
dist, _ = tree.query(prediction)
d2 = dist
gen_to_gt_chamfer = np.mean(dist)
gen_to_gt_chamfer_sq = np.mean(np.square(dist))

cd = gt_to_gen_chamfer + gen_to_gt_chamfer
```

B.4. Sampling algorithm

We show the sampling algorithm (Algorithm 2) proposed in EDM [22] for completeness.

Algorithm 2 Sampling

procedure SAMPLING($\mathbf{x}, t_{i \in \{0,\dots,N\}}$)
  $\mathbf{x}_0 = t_0 \mathbf{n}$ where $\mathbf{n} \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$
  for $i \in \{0, 1, \dots, N-1\}$ **do**
    $\mathbf{d}_i = (\mathbf{x}_i - D_\theta(\mathbf{x}_i, t_i))/t_i$
    $\mathbf{x}_{i+1} = \mathbf{x}_i + (t_{i+1} - t_i) \cdot \mathbf{d}_i$
  end for
end procedure

B.5. Networks

The forward passes of the middle blocks and final block (illustrated in Fig. 7) are implemented as follows:

```
# x: B x C
# t: B x C
# middle block
c = emb_mp_linear(t, gain=emb_gain) + 1
x = normalize(x)
res = x_pre_mp_linear(mp_silu(x))
res = mp_silu(res * c.to(y.dtype))
res = x_post_mp_linear(res)
x = mp_sum(x, res, t=0.3)
```

```
# x: B x C
# t: B x C
# final block
c = emb_mp_linear(t, gain=final_emb_gain) + 1
x = x_pre_mp_linear(mp_silu(normalize(x)))
x = mp_silu(x * c.to(y.dtype))
out = x_post_mp_linear(x, gain=final_out_gain)
```

B.6. Vector fields

The vector fields are coordinate-based networks which outputs vectors pointing towards to the surface. We use Libigl library [20] to process the data.

```
# p: B x 3
p = np.random.randn(B, 3)
v, f = igl.read_triangle_mesh(obj_path)
d, _, c = igl.point_mesh_squared_distance(p, v,
    f)
unsigned_distances = np.sqrt(d) # B
vectors = c - p # B x 3
```

B.7. Meshing with Blender

We utilize the Geometry Nodes from the software Blender as an alternative way to meshing (see Fig. 2). The setup can be found in Fig. 22. Note that we use the same parameters for all the objects. The meshing results can be further improved for different objects.

B.8. Color fields

We use the hashing grids proposed by Instant-NGP to implement the color field network in Fig. 11. The implementation is from the official github repository.

```
encoder_config = """{
    "otype": "HashGrid",
    "n_levels": 16,
    "n_features_per_level": 2,
    "log2_hashmap_size": 19,
    "base_resolution": 16,
}"""


network_config = """{
    "otype": "FullyFusedMLP",
    "activation": "ReLU",
    "output_activation": "Sigmoid",
    "n_neurons": 64,
    "n_hidden_layers": 2
}"""

class ColorField(nn.Module):
    def __init__(self):
        super().__init__()
```

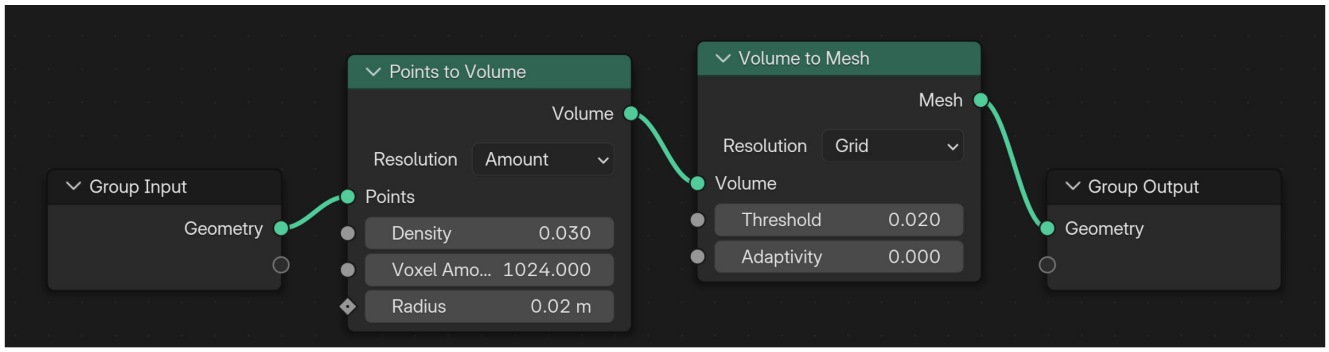

Figure 22. Meshing with Blender's Geometry Nodes

```
        self.encoding = tcnn.Encoding(
    n_input_dims=3, encoding_config=json.loads(
    encoder_config))
        self.network = tcnn.Network(
    n_input_dims=self.encoding.n_output_dims,
    n_output_dims=3, network_config=json.loads(
    network_config))

    def forward(self, x):
        x = self.encoding(x)
        x = self.network(x)
        return x
```

We optimize L_1-loss between the predicted and ground-truth colors. The training takes around 3 minutes. When the training is done, we can query colors for all spatial points,

$$\text{ColorField}(\mathbf{x}) = \mathbf{c}. \tag{B.1}$$

The colors in Fig. 11 are obtained by $\text{ColorField}(\mathcal{E}(\mathbf{n}))$ where $\mathbf{n} \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$.

B.9. Compression rate

Using geometry distributions to represent 3D surfaces offers several advantages. For example, at a given budgeted resolution, this representation provides natural sampling without computational overhead. Any number of surface points can be sampled directly from the geometry distribution to approximate the surface (see Fig. 6 for one example). As a result, it is no longer necessary to store extremely high-resolution point clouds to capture details. Instead, we can store the trained network, which theoretically retains all the information needed to recover the geometry, and sample surface points at the desired resolution for each use case. In Tab. B.2 we quantify the compression rate.

# Network blocks	2	4	6	8	10
# Parameters ($\times 10^6$)	2.38	3.96	5.53	7.11	8.68
Comp. ratio on 10^6 *points*	1.261	0.758	0.542	0.422	0.346
Comp. ratio on 10^9 *points*	1261	758	542	422	346

Table B.2. **Application: geometry compression.** We calculate the compression ratio on different numbers of sampled points (3 floats per point), assuming a network parameter is represented with one float. The chamfer distance to the ground-truth mesh is as in Tab. 2d. Since this method can represent an infinite number of points, the storage requirements remain constant regardless of the number of points. As a result, when representing a large number of points, the compression rate becomes more significant.

C. Discussions

C.1. Comparison with UDFs

While UDFs (unsigned distance fields) can model open surfaces and are of academic interest, they remain *niche* and *non-competitive* in practice due to three key limitations:

1. UDFs are inherently non-smooth (i.e. non-differentiable) at the zero-level set. See Fig. 23 *left* for a visualization of this issue. We hypothesize that neural networks are biased toward learning smooth functions, making non-smooth functions significantly harder to optimize.
2. UDFs exhibit gradient ambiguities at some points. The Marching Cubes algorithm for UDF meshing requires differentiation during inference, but these gradient ambiguities make the directional vectors used in the meshing algorithm ill-defined, preventing accurate conver-

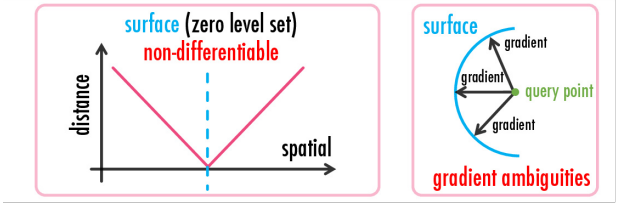

Figure 23. UDFs are non-differentiable at zero-level sets (*left*). The gradients are ill-defined at some points (*right*).

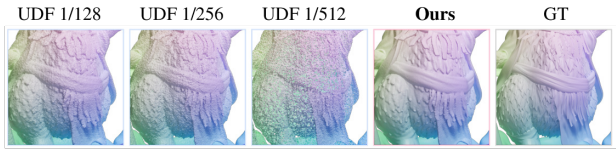

Figure 24. We compare our results with UDFs. The meshing outputs from UDFs are obtained using different ϵ-isosurfaces (1/128, 1/256, 1/512). While smaller ϵ are theoretically preferable, setting $\epsilon = 1/512$ introduces *numerous holes* in the mesh. Larger ϵ values (1/128, 1/256) also *fail to achieve smooth surfaces*, unlike our method, which delivers high-quality geometry.

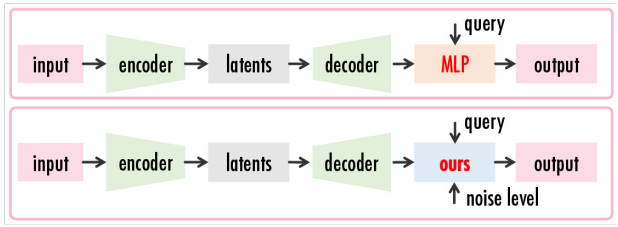

Figure 25. Template network design: MLP (*top*) *vs*. ours (*bottom*).

gence towards the ground-truth surface (see Fig. 23, *right*).

3. UDFs demonstrate inferior empirical performance. We validate this by training a UDF with InstantNGP and extracting meshes using DISO[1]. Even with this optimized pipeline, UDFs underperform compared to our method (see Fig. 24).

C.2. Generalizable case

Our method naturally extends to dataset encoding. During training, our network incorporates an additional noise-level parameter compared to conventional MLPs. See Fig. 25 for a template network design. During inference, the reconstruction process follows an iterative procedure akin to conditional diffusion.

[1] https://github.com/SarahWeiii/diso

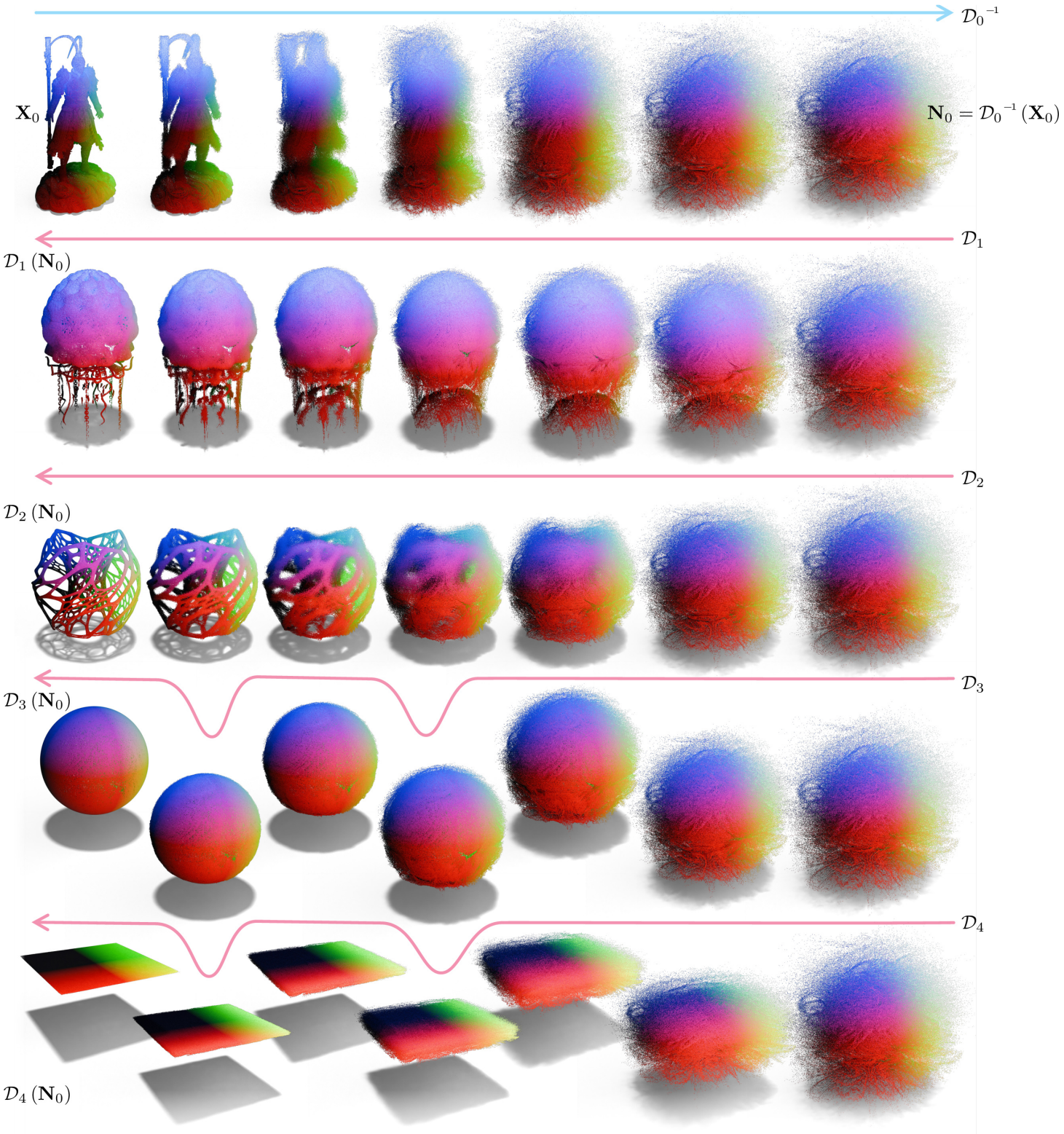

Figure 26. We denote the trained diffusion networks that map from a Gaussian distribution to the Wukong, jellyfish, lamp, sphere, and plane mesh as $\mathcal{D}_0, \mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4$, respectively. We sample 1 million points from the WuKong mesh, denoting these samples as $\mathbf{X}_0$. Applying the inversion we obtain $\mathbf{N}_0 = \mathcal{D}_0{}^{-1}(\mathbf{X}_0)$. In the leftmost column, we show the samples $D_1(\mathbf{N}_0), \mathcal{D}_2(\mathbf{N}_0), \mathcal{D}_3(\mathbf{N}_0), \mathcal{D}_4(\mathbf{N}_0)$, which closely approximate the original shapes, demonstrating the accuracy of our trained diffusion nets. For $D_0{}^{-1}$, we show results at timesteps $0, 10, 15, 20, 25, 30, 64$ aligned from left to right. For $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4$ we show results at timesteps $30, 40, 45, 48, 52, 55, 64$, aligned from right to left.