Geometry Distributions

Supplementary Material

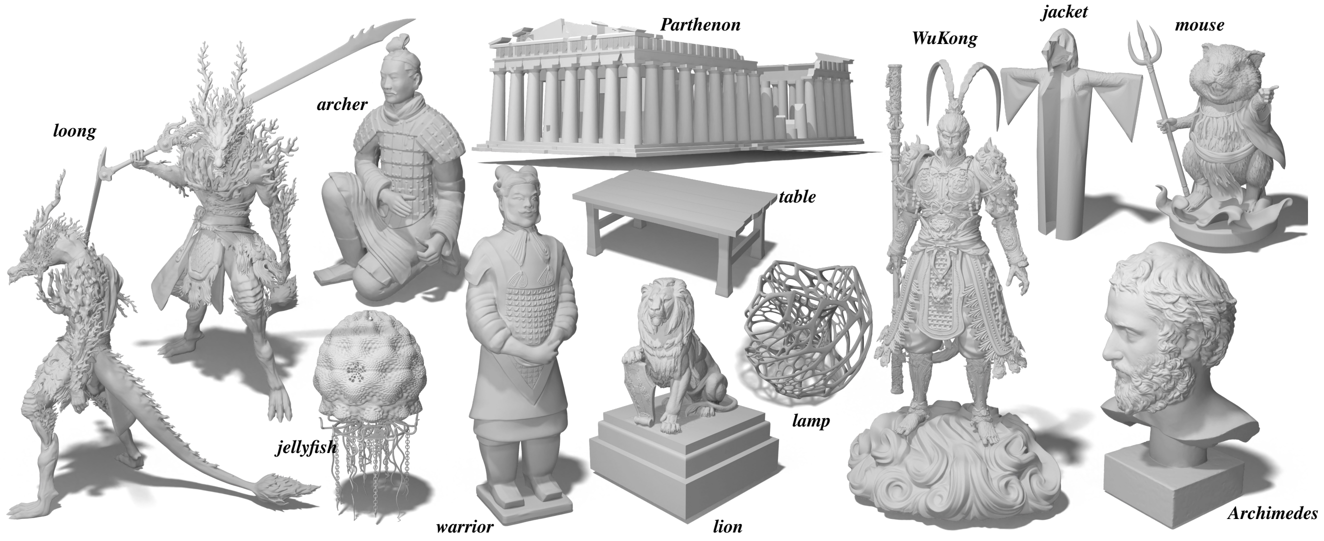


Figure 15. The ground-truth geometries for the shapes shown in Fig. 1.

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. 15.

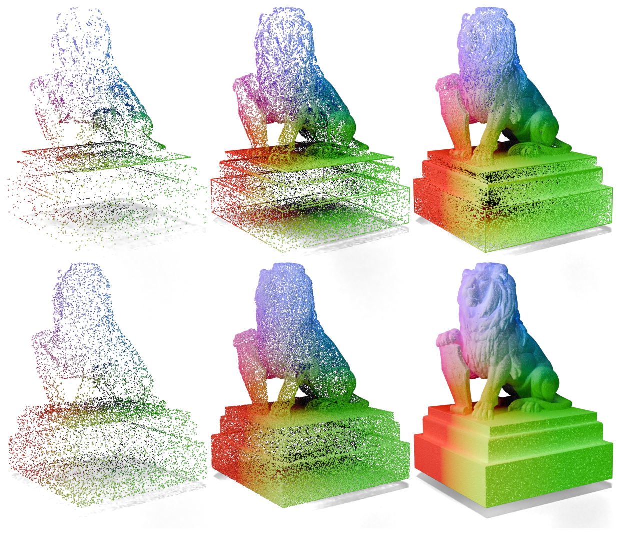


Figure 16. Additional results on comparison to vector fields-based method (extension of Fig. 2), our **GEOMDIST** produces more uniformly distributed samples with higher fidelity. *Top:* vector fields. *Bottom:* ours. From left to right, we show results of 10 thousands, 100 thousands and 1 million points.

A. Additional results

Fig. 16 extends Fig. 2, demonstrating that our approach generates more uniformly distributed samples with higher fidelity across different resolutions, compared to the vector field-based method. Fig. 18 (an extension of Tab. 3c), and Fig. 17 (an extension of Fig. 13), 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. 19 and Fig. 20, we show additional results of using our method to represent textured geometry and high-resolution scene. In Fig. 21 we show additional re-

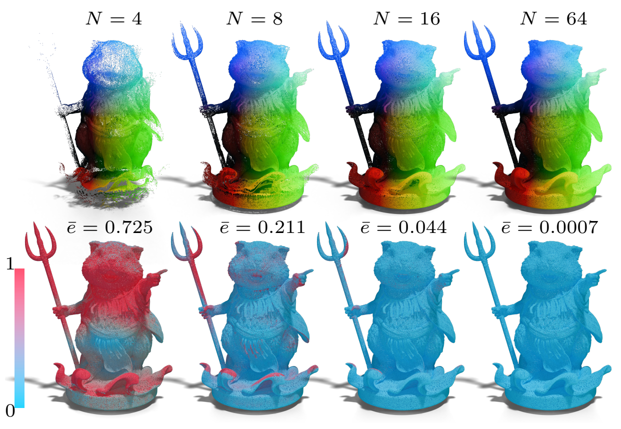


Figure 17. *Top*: the recovered shape from inversion $\mathcal{E} \circ \mathcal{D}(\mathbf{x})$ using different number of inverse sampling steps as in Eq. (4). *Bottom*: we color the per-sample error of the inversion on the original shape and report the average error $\bar{e}$ (L_2 distance to the surface).

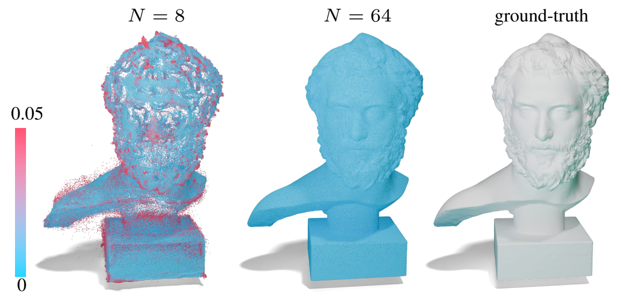


Figure 18. Errors (L_2 distance to the surface) of 1 million points using different number of sampling steps N as in Eq. (3). *Left*: $N = 8$. *Middle*: $N = 64$. *Right*: ground-truth mesh. The per-point color is visualized according to the error ($l2$ distance to the ground-truth surface). The results of $N = 16$ and $N = 32$ are very close as shown in Tab. 3c, thus are omitted here.

sults 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.

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.

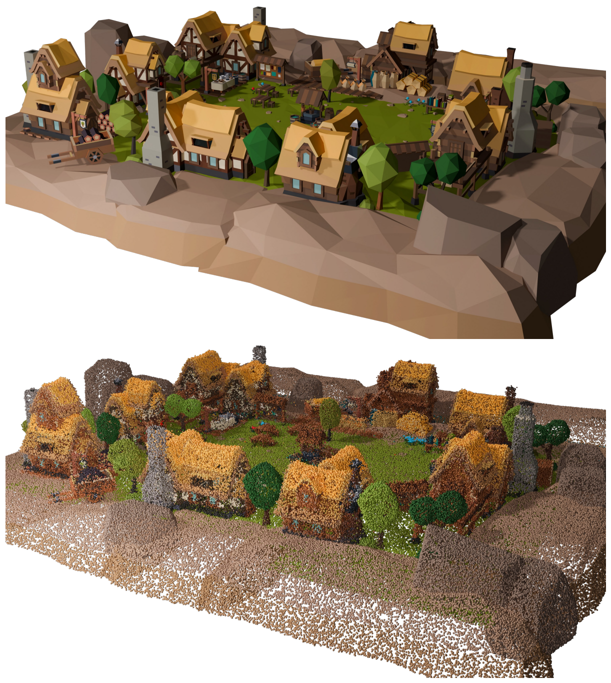


Figure 19. Applications on textured geometries. The setup is the same with Fig. 9. *Top:* ground-truth. *Bottom:* ours.

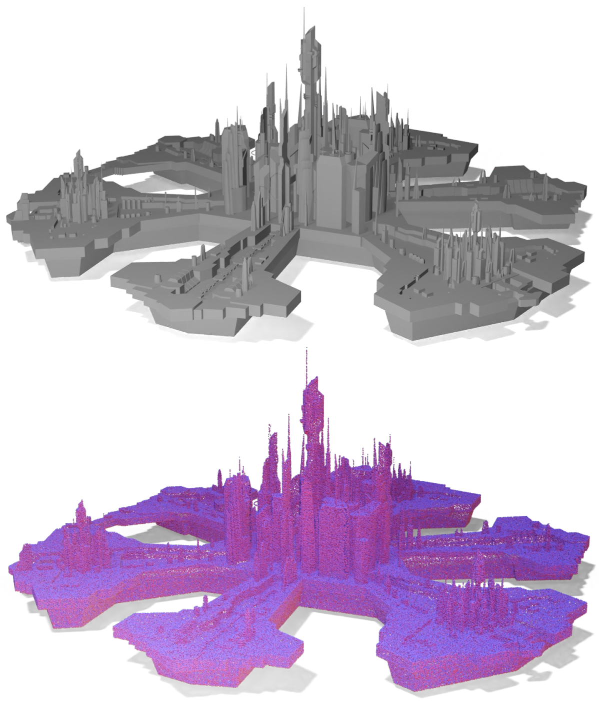


Figure 20. Results on city. *Top:* ground-truth. *Bottom:* ours.

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

B.2. Uniform distribution

When using uniform distribution as the initial noise in diffusion models (e.g., see Fig. 4), 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:

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

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 [19] for completeness.

Algorithm 2 Sampling

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

B.5. Networks

The forward passes of the middle blocks and final block (illustrated in Fig. 6) 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 [17] 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. Color fields

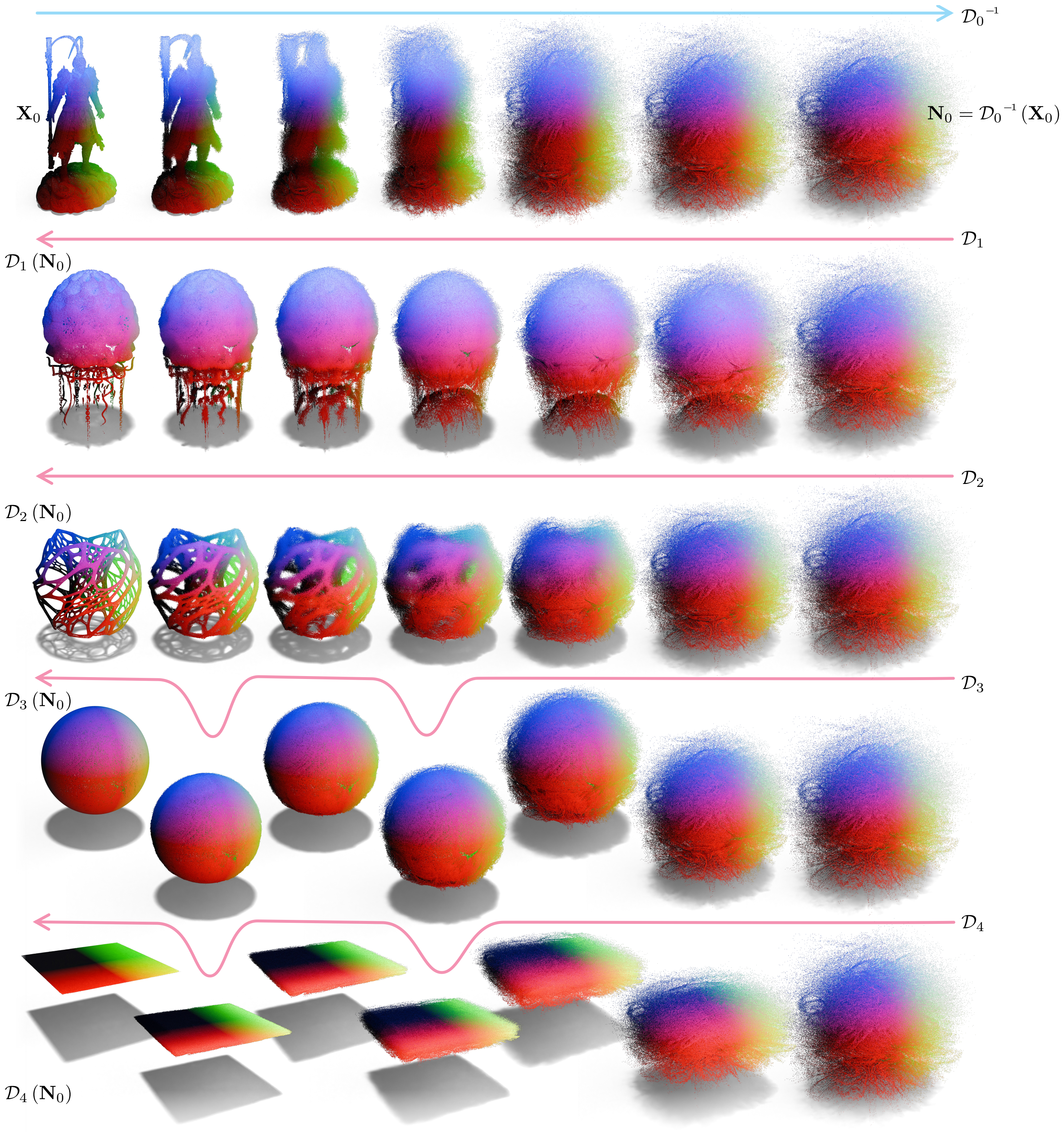
We use the hashing grids proposed by Instant-NGP to implement the color field network in Fig. 10. The implementation is from the official github repository.

```
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__()
        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,

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

The colors in Fig. 10 are obtained by $\mathrm{ColorField}(\mathcal{E}(\mathbf{n}))$ where $\mathbf{n} \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$.



$\mathcal{D}_0{}^{-1}$
$\mathbf{X}_0$
$\mathbf{N}_0 = \mathcal{D}_0{}^{-1}(\mathbf{X}_0)$
$\mathcal{D}_1$
$\mathcal{D}_1(\mathbf{N}_0)$
$\mathcal{D}_2$
$\mathcal{D}_2(\mathbf{N}_0)$
$\mathcal{D}_3$
$\mathcal{D}_3(\mathbf{N}_0)$
$\mathcal{D}_4$
$\mathcal{D}_4(\mathbf{N}_0)$

Figure 21. 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.