A LIMITATIONS AND FUTURE WORK

Figure 4: The failure cases generated by CDiffSDF.

Our framework predominantly yields two categories of generation failures. The first category entails the inability to accurately reconstruct the overall shape, typically manifesting as disjointed refactored components and artifacts, illustrated in the 1st and 2nd columns of Figure 4. The second category involves a mismatch between the reconstructed shapes and the input text prompt, highlighted in the 3rd column.

Mitigation of the first type of failure can potentially be achieved with an augmentation of training data. This enhancement allows the diffusion model to more precisely approximate the latent code distribution and enables the latent code-based SDF model to more robustly reconstruct shapes from the latent codes. Addressing the second category of failures necessitates the development of advanced techniques to enhance the attention of diffusion models to conditions. An increase in shape-text pairs could also contribute to the alleviation of this issue.

Echoing the requirements of (Chen et al., 2018; Liu et al., 2022; Li et al., 2023), CDiffSDF necessitates sizable 3D-text pairs for model training. This demand underscores the need for the community to meticulously annotate 3D shapes through crowdsourcing to amass larger datasets, albeit a costly endeavor. A recent advancement involves utilizing pre-trained captioning models and Large Language Models (LLM) to annotate expansive 3D datasets (Luo et al., 2023b; Deitke et al., 2023). Another promising approach entails the recovery of 3D shapes (Gkioxari et al., 2022; Qian et al., 2022) from 2D/Video-Text pairs, subsequently yielding 3D-text pairs.

Moreover, here is no assured guarantee that the proposed framework can seamlessly scale up with web-scale 3D-text pairs given the adoption of a limited-size encoder and diffusion model. Besides, a crucial future development involves the expansion of our framework to incorporate texture, while maintaining its lightweight characteristics.

B VOXEL-BASED SDF TEXT-TO-SHAPE

As mentioned in Section 4, we have studied voxel-like SDF representation (Shen et al., 2021) before moving into the latent-code-based SDF formulation. This section shares some of our discoveries. For the voxel-like SDF representation, we refer to a voxel $32 \times 32 \times 32$, where each cell stores the corresponding SDF value, as shown in the left of Figure 5. We can reconstruct the mesh based on the voxel representation via marching cubes as shown in the right of Figure 5.

Accordingly, we used a diffusion model that can consume the voxel-like SDF representation, where we adopt a U-Net structure with three-levels of downsampling and upsampling, 3D convolutional layers, and Group Normalization layers (Wu & He, 2018). We follow the same training setting as we used in our latent-code-based SDF diffusion models, listing in Appendix F.1.

However, our results show the voxel-like SDF + 3D UNet diffusion models failed to capture the 3D object distribution, as shown in Figure 6. The diffusion inference of the voxel-like SDF representation costs 110.38 seconds for sampling a 48 batch which significantly slower than the latent-code-based SDF solution which cost 15.21 seconds. Therefore, we do not further play with the voxel-based solution and delve deep in the latent-code-based SDF representation.

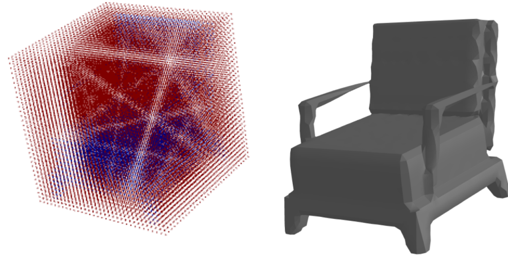

Figure 5: An instance of voxel-like SDF representation. Blue points mean inside the shape, and red points mean outside the shape. The right mesh is the reconstruction mesh on the left voxel SDF. Visualization is achieved via Pyrender with 'use_raymond_lighting == True'.

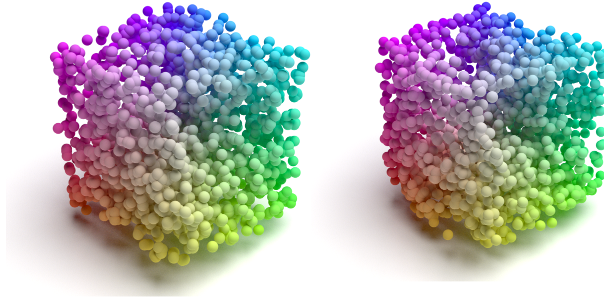

Figure 6: The Voxel-like SDF representation + 3D UNet diffusion model failed to capture the distribution of 3D shapes.

C MORE RESULTS

This section provides more results generated by *CDiffSDF*.

Figure 7: More results generated by CDiffSDF. The second row shows many different details contained in shapes that reflect the same text prompt.

Furthermore, we check whether CDiffSDF shows combinatorial generalizability that can generate new shape structures by composing known structures. We check this by looking at the nearest neighbors (Chamfer Distance) of the generated shape in our training data. Results are listed in Figure 10 and show our framework generates novel structures beyond training shapes to some extent.

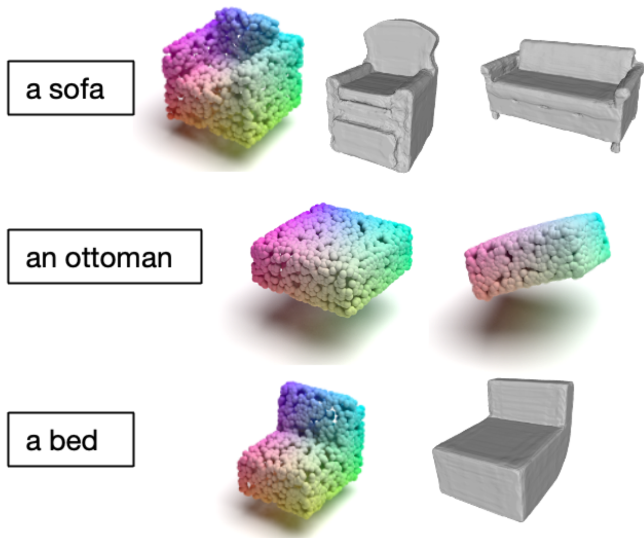

Figure 8: More results generated by CDiffSDF.

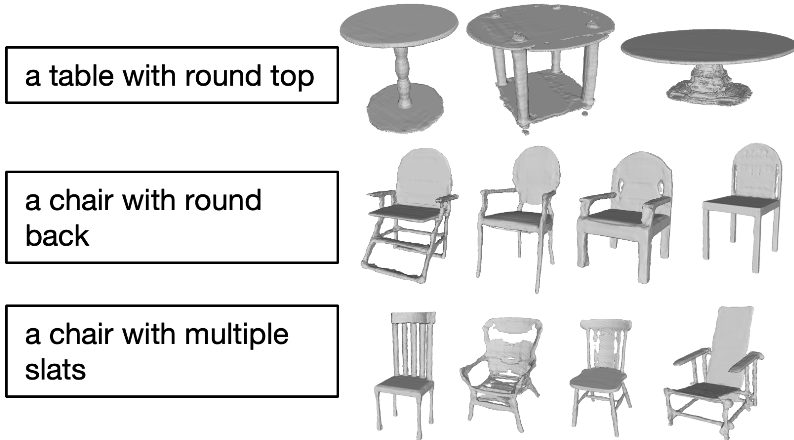

Figure 9: More results generated by CDiffSDF.

D SHAPE RECONSTRUCTION FROM LATENT CODES

For our latent-code-based SDF framework (Figure 11), we adopt the same 8 fully connected layers as (Park et al., 2019; Duan et al., 2020), but no latent connections, no Dropout layers, and the input dimensions change from $259 = 256 + 3$ to $296 = 256 + 30$, where we adopt $L = 10$ to positional encode three-dimensional coordinates. Following (Park et al., 2019), we sampled 500,000 points from each shape before training, and sampled $\mathcal{K}_{train}$ = 16,384 points during each training optimization, and $\mathcal{K}_{sample}$ = 8,000 in each inference optimization. We trained all the SDF models for 2,000 epochs with batch size 96, learning rate $1e - 3$ for the latent codes optimization, and learning rate $5e - 4$ for the model optimization. The code regularization term is set to $\omega = 1e - 4$. We add Gaussian noise with mean 0 and std 0.05 during training, as mentioned in Section 3.3. Also, unlike (Duan et al., 2020), which adjusted λ by manually setting a series of intervals, we set up a continuous sigmoid-like schedule to gradually increase λ from 0 to 0.5: $0.5 * \left(1 - 1/1 + \exp\left(\left(e - \frac{\varepsilon}{2}\right) / \frac{\varepsilon}{20}\right)\right)$, where exp stands for exponential, e denotes the current epoch number, and ε denotes the total training epochs.

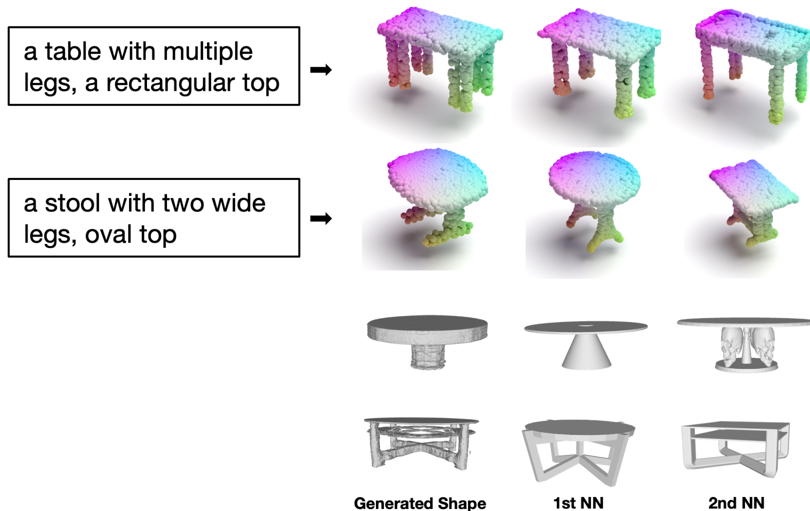

Figure 10: We inspect the top-2 nearest neighbors of the generated shapes in training datasets to check if CDiffSDF can compose known structures to generate new ones. The first two rows are text-to-shape generation, and the bottom two rows are unconditional generation.

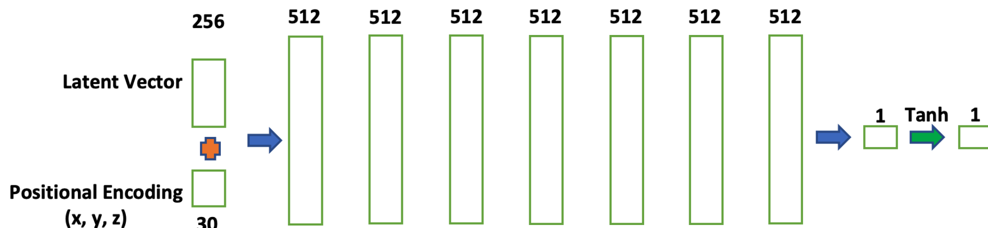

Figure 11: Network architecture illustration for our latent-code-based SDF model. All layers are connected via fully connected layers except the green arrow represents the Tanh activation function.

The rightmost column of Table 4 shows the results where we leave a subset of training shapes as validation dataset and tested the average Chamfer Distance between the reconstructed shape and its corresponding ground-truth one.

	PosEncoding	Latent	Dropout	Top-k	Weight	Score ↓
DeepSDF	✗	✗	✗	✗	✗	0.175
-	✗	✗	✗	✓	✓	0.137
-	✓	✗	✗	✗	✗	0.152
-	✓	✓	✗	✗	✗	0.15
-	✓	✓	✓	✗	✗	0.143
-	✓	✓	✓	✓	✗	0.119
Ours	✓	✓	✓	✓	✓	0.113

Table 4: **SDF model ablation studies**: numbers are mean CD with $\times 10^3$, tested in normalized validation sets.

Besides, we conducted further ablation studies on the selection of L which controls the length of our positional encoding, and $\boldsymbol{k}$ which controls the hard example mining magnitude. In our main experiments, we adopt $L = 10$ and $k = 0.5$.

	Score ↓
Ours-$k=0.3$	0.117
Ours-$k=0.7$	0.126
Ours-$L=1$	0.128
Ours-$L=30$	0.119
Ours	0.113

Table 5: **SDF model more ablation studies:** numbers are mean CD with $\times 10^3$, tested in normalized validation sets.

E DATASET

Here we provide some examples around the dataset (Luo et al., 2022) we used. It consist of shapes from Text2Shape (Chen et al., 2018), ABO (Collins et al., 2021), and ShapeGlot (Achlioptas et al., 2019) datasets. All the shapes in the dataset are furniture, and the main parts of them are chairs and tables, which contain a lot of structure variations.

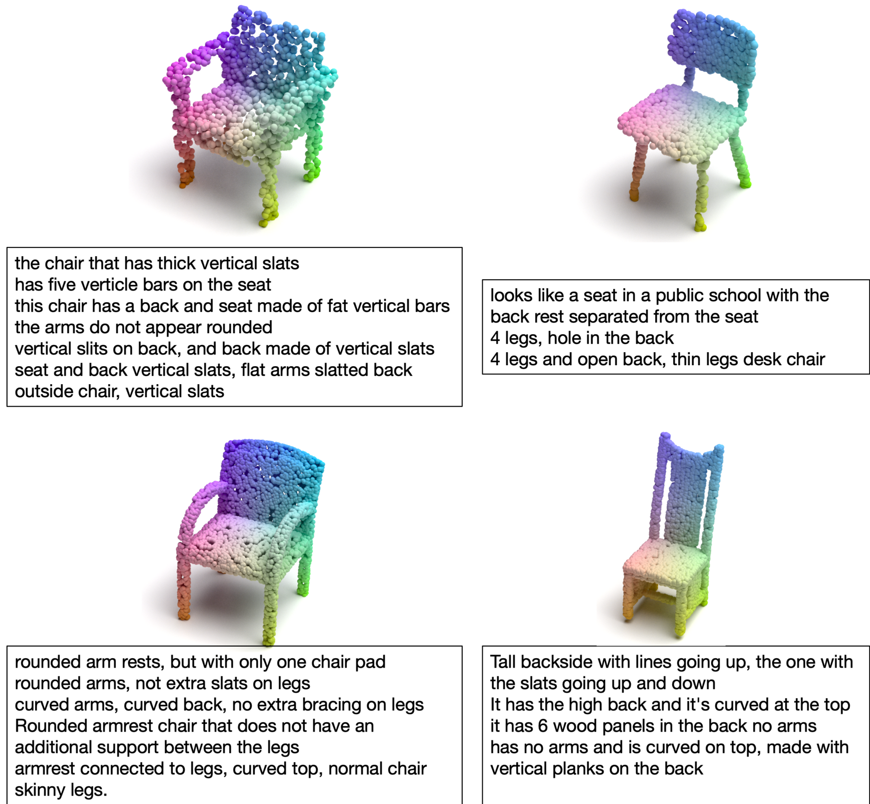

Figure 12: Random examples of shapes from ShapeGlot (Achlioptas et al., 2019). One description per sentence.

F IMPLEMENTATION DETAILS

This section lists model and training details around the proposed framework and the compared baselines. We also list examples of the adopted data, (Shape, Structure-Related Text), to provide some sense for our problem setting.

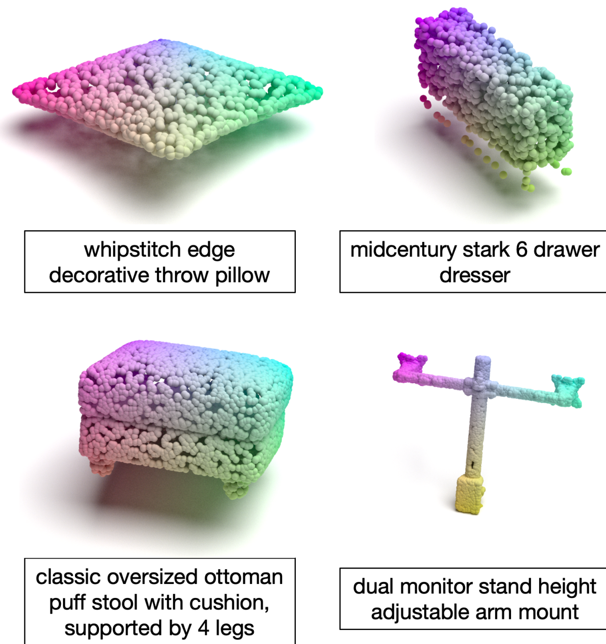

Figure 13: Random examples of shapes from ABO (Collins et al., 2021). One description per sentence.

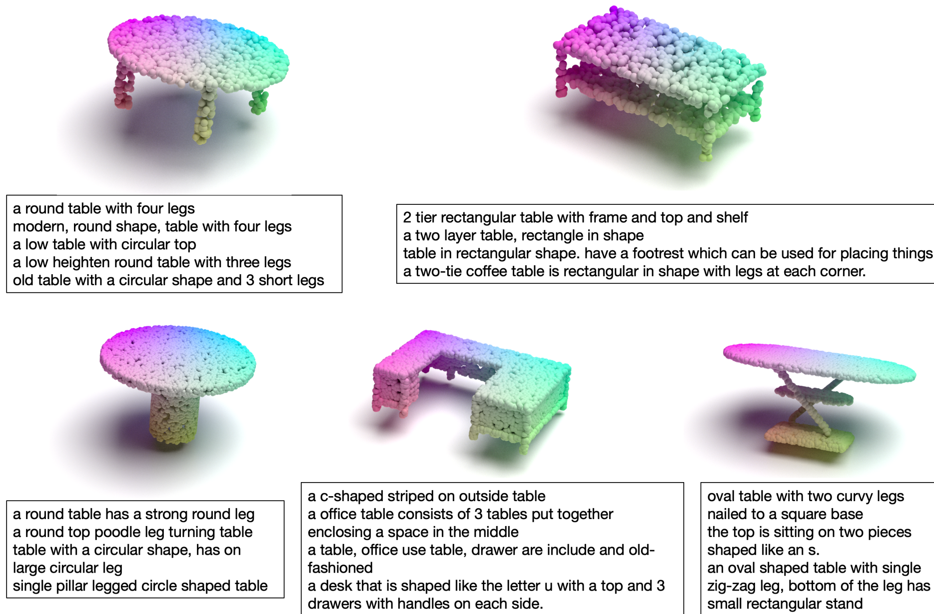

Figure 14: Random examples of shapes from Text2Shape (Chen et al., 2018). One description per sentence.

F.1 CDiffSDF

We adopt a decoder-only transformer (Vaswani et al., 2017) with casual attention (Child et al., 2019; Ramesh et al., 2022) as the diffusion model. The Transformer has depth 3, MLP wide 64, and 8 attention heads. We train the diffusion model with batch size 256, learning rate starting from $1e-3$ following by a cosine annealing schedule, the ADAM optimizer with default parameters, and a total of 10,000 epochs. We used a linear schedule to regulate β_t and compute α_t and $\bar{\alpha}_t$ accordingly.

We set $\beta_{t_0} = 0.004$ and $\beta_{t_{1000}} = 0.00002$ decided by the noise level used in training latent-code-based SDF model. For classifier-free guidance (Ho & Salimans, 2022), we used drop ratio 0.1. The diffusion training and sampling objective are listed in Section 3.2.

For visualizations used in this paper, we adopt Misuba (Jakob et al., 2022) to render point clouds and meshlab (Cignoni et al., 2008) to render meshes.

For other types of conditions, the general approach mirrors that of Text-to-3D conversion. Initially, input conditions are converted into embeddings, denoted as $\mathcal{B}_i$. These embeddings act as conditions in our learned diffusion model to predict a latent code, $\boldsymbol{z}_i$. In the case of image-conditioned scenarios, a frozen CLIP image encoder is utilized. For class-conditioned scenarios, one-hot encoding is employed for all classes. In unconditional situations, the condition input is omitted, directly applying the diffusion model to the latent codes of 3D objects.

F.2 Baseline Details

We compared with various Text-to-Shape baselines to examine the performance of our proposed *CDiffSDF*. For each baseline, we generally follow the default configurations provided by their Github, including learning rates, batch size, training epochs, and other hyper-parameters. The major modification we made is the data part, where different methods adopt different ways to process 3D objects and text. We list some of our modifications below:

- **Shape Compiler (Luo et al., 2022)**: We compare with Shape Compiler Limited listed in (Luo et al., 2022), due to we only use 3D-Text data and perform the Text-to-Shape task solely.
- **CLIP-Forge (Sanghi et al., 2022) and Dream Field (Jain et al., 2021)**: We follow their papers and use *clip.tokenize* and *clip.encode_text* to encode text. We used the pre-trained model provided in CLIP-Forge's Github to generate multiple shapes given one text prompt. We adopt the Pytorch implementation of Dream Field (Jain et al., 2021) for qualitative comparisons as shown in Figure 3. We do not adopt the provided codes in (Jain et al., 2021) Github because it takes more than eight days to process one single text prompt with our 4 GPUs of 2,048 Gb memory. Although, the Pytorch implementation still costs more than 10,000 seconds for a text prompt. Therefore, it is impractical to test Dream Feild in our test set, which consists of 15,000 text-shape pairs.
- **Shape IMLE (Liu et al., 2022)**: we process all text with Bert (Devlin et al., 2018) follow their codes (Liu et al., 2022). We adopt *BertTokenizer* and of the pre-trained flag "bert-base-uncased" to tokenize the text and then obtain the embedding by the Bert model with the same pre-trained flag.
- **CWGAN (Chen et al., 2018)**: we follow the data templates provided in (Chen et al., 2018) to process our data. We process each data entry as in (Chen et al., 2018), including *caption_tuples*, *caption_matches*, *vocab_size*, *max_caption_length*, and *dataset_size*. For the text tokenizer, we used spaCy (Honnibal et al., 2020) as mentioned in (Chen et al., 2018). (Chen et al., 2018) set the token number in a linear increase rule based on their data order, while we set our token number based on our data order. We abandoned descriptions with more than 96 tokens as (Chen et al., 2018) did.
- **CurrSDF (Duan et al., 2020) and DeepSDF (Duan et al., 2020)**: we follow their GitHub codes to train and implement without any code changes. The differences between CurrSDF (Duan et al., 2020) and DeepSDF (Park et al., 2019) are summarized in Table 1 of (Duan et al., 2020), where they manually set up a lot of stages to gradually change network structures and loss weights.
- **DreamFusion (Poole et al., 2022)**: we adopted the implementation provided by https://github.com/ashawkey/stable-dreamfusion. We use the plain NeRF implementation instead of Instant-NGP.
- **VoxelDiffSDF (Li et al., 2023)**: we directly follow the instructions provided by their office code release.
- **Point-E (Nichol et al., 2022) Shap-E (Jun & Nichol, 2023)**: we used the pre-trained model provided by OpenAI Githubs. For Shap-E, we reconstruct the output mesh via STF mode.