

Appendix

A. Method and network architectures

A.1. Sequential edge-vertex representation

In this section, we elaborate on the details of our method for serializing the edge-vertex adjacency relationships in the B-rep dataset during the training phase. Following the notations introduced in the main paper, we present our approach using the B-rep model \mathcal{B}_i as an example. The B-rep model \mathcal{B}_i consists of N_f^i faces, indexed as $0, 1, \dots, N_f^i - 1$. Each face contains a set of edges, denoted as \mathbf{FE}_i , which is the dual representation of the edge-face relationship \mathbf{EF}_i . To ensure unambiguous edge-to-edge connectivity, we introduce a unique indexing scheme for edge endpoints. Specifically, for the j -th edge e_j^i , we assign IDs $2j$ and $2j+1$ to its two endpoints, referred to as even and odd endpoints respectively. In the main paper, we refer to this process as edge duplication, which we will leverage in Appendix A.3. Consequently, the B-rep model \mathcal{B}_i contains $2N_e^i$ unique endpoint IDs. To handle the many-to-one correspondence between endpoints and vertices, we define a mapping vector $\tilde{\mathbf{V}}_i$ of length $2N_e^i$, where $\tilde{\mathbf{V}}_i[j]$ stores the vertex ID in \mathbf{V}_i corresponding to the j -th endpoint. The mapping vector is initialized with -1 values and updated progressively during computation.

The serialization process starts by initializing an empty sequence \mathbf{EV}_i^{seq} . For each face \mathbf{f}_k^i beginning with \mathbf{f}_0^i , we first identify the edge with the minimum ID as our starting point and add its even endpoint ID to \mathbf{EV}_i^{seq} . Following the loop structure, we iteratively process connected edges by selecting the connecting edge with the smaller ID at each step and adding its corresponding endpoint ID to \mathbf{EV}_i^{seq} . When a loop is completed, we append a loop-end flag e_{loop} (numerically represented as -1). This process repeats for any remaining edges within the face, treating each as a new starting point for subsequent loops. After processing all edges in a face, we append a face-end flag e_{face} (numerically represented as -2) before moving to the next face. Fig. 1 illustrates our approach with a simple example: a loop composed of edges e_2 , e_4 , e_7 , and e_5 . The endpoint IDs are labeled adjacent to their corresponding vertices (e.g., endpoint IDs 4 and 10 map to the same vertex). Following our serialization procedure, this loop generates the sequence $(4, 8, 15, 11, -1)$. The complete algorithm is formally presented in Algorithm 1.

A.2. Details of edge-face adjacency generation

Graph-sequence representation. Fig. 2 provides a simple example illustrating three topologically equivalent representations of edge-face relationships in a B-rep: (1) the edge-face graph, (2) the face-edge-face matrix, and (3) the edge-face sequence. The example is based on a B-rep with

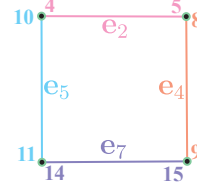


Figure 1. Illustration of our sequential edge-vertex representation. A simple loop consisting of four edges (e_2 , e_4 , e_7 , and e_5) is shown with their endpoint IDs labeled. The loop is serialized into sequence $(4, 8, 15, 11, -1)$.

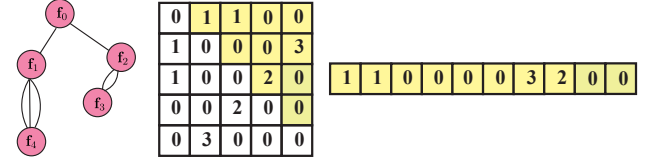


Figure 2. An illustrative example of three topologically equivalent representations of the edge-face relationships in a B-rep: (1) an edge-face graph, where nodes represent faces and connections represent shared edges; (2) a face-edge-face matrix, where entries indicate the number of shared edges between two faces; and (3) a serialized edge-face sequence, obtained by flattening the upper triangular part of the matrix in row-major order.

five faces, where pairs of faces share zero, one, or multiple edges. As described in the main paper, the face-edge-face matrix encodes the number of shared edges between any two faces in the model. A matrix entry of 0 indicates no shared edges between the corresponding pair of faces. Exploiting the symmetry of this matrix, we restrict our attention to its upper triangular part (excluding the diagonal) for further processing. The upper triangular elements are then serialized into a sequence by flattening them in row-major order, ensuring that the sequence retains the complete topological adjacency information of the faces. Our Edge-Face Adjacency generation model \mathcal{E}_θ^{EF} is designed to learn the distribution of these sequences, effectively capturing the underlying topological patterns present in the edge-face relationships of B-rep models. This sequential representation simplifies the learning task while retaining all necessary adjacency information.

Embedding. To standardize the varying lengths of sequences $\{\mathbf{EF}_i^{seq}\}_{i=1}^N$, we pad each face-edge-face matrix \mathbf{FEF}_i to a fixed size $M_f \times M_f$ using zeros, where M_f is the maximum number of faces across $\{\mathcal{B}_i\}_{i=1}^N$. The edge-face sequence is then extracted from the upper triangular part of the padded matrix in a row-major order. For simplicity, we continue to denote the padded matrix and the extracted sequence as \mathbf{FEF}_i and \mathbf{EF}_i^{seq} , respectively. Note that the length of the serialized edge-face sequence \mathbf{EF}_i^{seq} is $\frac{M_f(M_f-1)}{2}$. We define the embedding function in the edge-

Algorithm 1 Sequential Edge-Vertex Representation

Require: Face-Edge adjacency matrix \mathbf{FE}_i , Edge-Vertex adjacency matrix \mathbf{EV}_i

Ensure: Edge-Vertex sequence \mathbf{EV}_i^{seq}

```
1: function OPP_VERT( $\mathbf{v}, \mathbf{e}$ )  $\triangleright$  Returns the vertex ID  
   opposite to  $\mathbf{v}$  in edge  $\mathbf{e}$   
2: end function  
3: function OPP_ENDPNT( $\mathbf{ep}$ )  $\triangleright$  Returns the endpoint ID  
   opposite to  $\mathbf{ep}$   
4: end function  
5: Initialize number of faces  $N_f^i$  and number of edges  $N_e^i$   
6: Initialize  $\mathbf{EV}_i^{seq} \leftarrow \emptyset$   
7: Initialize  $\mathbf{e}_{loop} \leftarrow -1, \mathbf{e}_{face} \leftarrow -2$   
8: Initialize  $\tilde{\mathbf{V}}_i \leftarrow \{-1\}^{2N_e^i}$   $\triangleright$  Mapping vector  
9: for face index  $j \leftarrow 0$  to  $N_f^i - 1$  do  
10:    $\mathbf{E}_{rest} \leftarrow \mathbf{FE}_i[j]$   $\triangleright$  Edges of the current face  
11:   while  $\mathbf{E}_{rest} \neq \emptyset$  do  
12:      $\mathbf{e}_{cur} \leftarrow \min(\mathbf{E}_{rest})$   $\triangleright$  Starting edge  
13:     Remove  $\mathbf{e}_{cur}$  from  $\mathbf{E}_{rest}$   
14:      $\mathbf{ep} \leftarrow 2\mathbf{e}_{cur}$   $\triangleright$  Current endpoint  
15:     Append  $\mathbf{ep}$  to  $\mathbf{EV}_i^{seq}$   
16:      $\mathbf{v}_{cur} \leftarrow$  corresponding vertex of  $\mathbf{ep}$   
17:      $\mathbf{v}_{opp} \leftarrow \text{opp\_vert}(\mathbf{v}_{cur}, \mathbf{e}_{cur})$   
18:      $\tilde{\mathbf{V}}_i[\mathbf{ep}] \leftarrow \mathbf{v}_{cur}$   
19:      $\tilde{\mathbf{V}}_i[\text{opp\_endpnt}(\mathbf{ep})] \leftarrow \mathbf{v}_{opp}$   
20:      $\mathbf{v}_{cur} \leftarrow \mathbf{v}_{opp}$   
21:     while True do  $\triangleright$  Process current loop  
22:       Find  $\mathbf{e}_{next} \in \mathbf{E}_{rest}$  connected to  $\mathbf{v}_{cur}$   
23:       if  $\mathbf{e}_{next} = \mathbf{e}_{start}$  then  $\triangleright$  Loop closed  
24:         Append  $\mathbf{e}_{loop}$  to  $\mathbf{EV}_i^{seq}$   
25:         break  
26:       else  
27:          $\mathbf{ep} \leftarrow$  corresponding endpoint to  $\mathbf{v}_{cur}$   
28:         Append  $\mathbf{ep}$  to  $\mathbf{EV}_i^{seq}$   
29:          $\mathbf{v}_{opp} \leftarrow \text{opp\_vert}(\mathbf{v}_{cur}, \mathbf{e}_{next})$   
30:          $\tilde{\mathbf{V}}_i[\mathbf{ep}] \leftarrow \mathbf{v}_{cur}$   
31:          $\tilde{\mathbf{V}}_i[\text{opp\_endpnt}(\mathbf{ep})] \leftarrow \mathbf{v}_{opp}$   
32:         Remove  $\mathbf{e}_{next}$  from  $\mathbf{E}_{rest}$   
33:          $\mathbf{e}_{cur} \leftarrow \mathbf{e}_{next}$   
34:          $\mathbf{v}_{cur} \leftarrow \mathbf{v}_{opp}$   
35:       end if  
36:     end while  
37:   end while  
38:   Append  $\mathbf{e}_{face}$  to  $\mathbf{EV}_i^{seq}$   $\triangleright$  Mark end of the face  
39: end for  
   return  $\mathbf{EV}_i^{seq}$ 
```

face model as:

$$EM_{\theta}^{EF} : \{0, 1, \dots, M_e\}^{\frac{M_f(M_f-1)}{2}} \rightarrow \mathbb{R}^{\frac{M_f(M_f-1)}{2} \times d_{ef}}, \quad (1)$$

where this function maps each integer in \mathbf{EF}_i^{seq} to a d_{ef} -

dimensional embedding, capturing various semantic and structural properties. Let SE_{θ} , POS_{θ} , and FID_{θ} represent the shared-edges embedding, positional embedding, and face ID embedding, respectively. The final embedding is computed as:

$$EM_{\theta}^{EF}(\mathbf{EF}_i^{seq}) = SE_{\theta}(\mathbf{EF}_i^{seq}) + POS_{\theta}(\mathbf{EF}_i^{seq}) + FID_{\theta}(\mathbf{EF}_i^{seq}), \quad (2)$$

where the shared-edges embedding is defined as:

$$SE_{\theta}(\mathbf{EF}_i^{seq}) = (W_{\theta} \cdot \text{onehot}(\mathbf{EF}_i^{seq}))^T, \quad (3)$$

Here, $W_{\theta} \in \mathbb{R}^{d_{ef} \times (M_e+1)}$ is a learnable matrix and $\text{onehot}(\mathbf{EF}_i^{seq}) \in \{0, 1\}^{(M_e+1) \times \frac{M_f(M_f-1)}{2}}$ represents the one-hot encoding of \mathbf{EF}_i^{seq} . The positional embedding POS_{θ} indicates the position index of each token in the sequence, implemented using the sinusoidal positional encoding scheme proposed in [7]. For the face ID embedding, we assign each face a learnable d_{ef} -dimensional embedding vector. The embedding of an edge is computed as the average of the embeddings of the two faces it connects. Formally, for the k -th element in the sequence \mathbf{EF}_i^{seq} , the corresponding face ID embedding $FID_{\theta}(\mathbf{EF}_i^{seq})[k]$ is defined as:

$$FID_{\theta}(\mathbf{EF}_i^{seq})[k] = \frac{(V_{\theta} \cdot \text{onehot}(k_{row}))^T}{2} + \frac{(V_{\theta} \cdot \text{onehot}(k_{col}))^T}{2}, \quad (4)$$

where $V_{\theta} \in \mathbb{R}^{d_{ef} \times M_f}$ is a learnable matrix, and k_{row}, k_{col} represent the row and column indices of the k -th element in \mathbf{FeF}_i 's upper triangular portion.

Training workflow. During training, the sequence \mathbf{EF}_i^{seq} is fed into the Transformer encoder, which outputs a contextual embedding with shape $\frac{M_f(M_f-1)}{2} \times d_{ef}$. To obtain the latent code, we compute the mean of the embeddings for all tokens in the encoder's output. This aggregation step ensures that the latent code encapsulates global contextual information from the input sequence. In the decoding phase, a special token is prepended to the sequence \mathbf{EF}_i^{seq} as a "begin" token, while the last token of the sequence is removed. The modified sequence is then passed through the embedding layer, as discussed in the previous paragraph, to obtain its embeddings. These embeddings are combined with the latent code and fed into the masked Transformer decoder, which outputs a probability distribution over all possible tokens for each position in the sequence. This architecture aligns with the framework of a standard Variational Autoencoder (VAE) [4], where the encoder and decoder are jointly optimized. Tab. 1 summarizes the output shapes of each module during the training phase.

Autoregressive generation. During inference, we sample a latent code from the learned latent space and initialize the sequence with a “begin” token. The initialized sequence, combined with the latent code, is fed into the Transformer decoder to obtain the probability distribution of the first token. We sample from this distribution to generate the first token of the sequence. Subsequently, this sampled token is appended to the current sequence, which is then passed through the decoder to predict the distribution of the next token. This process is repeated iteratively until the sequence reaches the predefined length (*i.e.*, $\frac{M_f(M_f-1)}{2}$).

A.3. Details of edge-vertex adjacency generation

Face feature extraction. To integrate the information from the edge-face adjacency (edge-face graph) generated in the previous step, we employ a graph convolutional network (GCN) [5]. Specifically, the feature of each face node is initialized based on the number of edges it contains, while the weight of each edge between two connected face nodes is determined by the number of edges they share (\mathbf{FeF}_i/M_e). These initial node features and edge weights are input into a two-layer GCN, which aggregates and transforms information from neighboring nodes. This process yields a final embedding matrix with shape $N_f^i \times d_{ev}$, where d_{ev} is the dimensionality of the aggregated features for each face. The resulting face node embeddings are then combined with the face ID embeddings to form the final aggregated face features. It is worth noting that this face feature extraction step is optional, as it may slightly increase training time while offering marginal performance gains.

Endpoint embedding. To generate the endpoint embeddings, each edge in the \mathbf{E}_i is duplicated, producing two copies corresponding to its two endpoints, as illustrated in Fig. 3. The resulting duplicated edges is denoted as

$$\tilde{\mathbf{E}}_i = \{\mathbf{e}_0^{i,even}, \mathbf{e}_0^{i,odd}, \mathbf{e}_1^{i,even}, \dots, \mathbf{e}_{N_e^i-1}^{i,odd}\}, \quad (5)$$

To encode the endpoint-specific information, we introduce two learnable vectors, $emb_{even} \in \mathbb{R}^{d_{ev}}$ and $emb_{odd} \in \mathbb{R}^{d_{ev}}$, representing the embeddings for “even-indexed edges” and “odd-indexed edges”, respectively. The final endpoint embedding for the edge set $\tilde{\mathbf{E}}_i$ is constructed as a sequence of these embeddings, defined as

$$EDP(\tilde{\mathbf{E}}_i) = \{emb_{even}, emb_{odd}, \dots, emb_{even}, emb_{odd}\}, \quad (6)$$

where $|EDP(\tilde{\mathbf{E}}_i)| = 2N_e^i$. This structured embedding provides a consistent and learnable representation for edge endpoints, enabling efficient feature encoding for embedding tasks.

Training workflow. During the training phase, the edges \mathbf{E}_i together with two special tokens \mathbf{e}_{loop} and \mathbf{e}_{face} are embedded into $2N_e^i + 2$ vectors, each with dimension

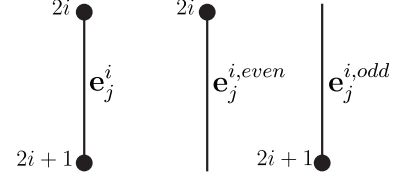


Figure 3. Each edge in the \mathbf{E}_i is duplicated to form $\tilde{\mathbf{E}}_i$, with even- and odd-indexed edges assigned to represent its two endpoints.

d_{ev} . These embeddings are processed by the Transformer encoder to obtain contextual embeddings $\mathcal{E}_\theta^{EV}(\mathbf{E}_i) \in \mathbb{R}^{(2N_e^i+2) \times d_{ev}}$. For the decoding process, each integer in the ground-truth edge-vertex sequence \mathbf{EV}_i^{seq} is mapped to its corresponding contextual embedding, maintaining the edge connectivity order. These aligned embeddings, combined with the positional embedding POS_θ , are then processed by a masked Transformer decoder followed by a pointer network to produce decoder embeddings $\mathcal{D}_\theta^{EV}(\mathcal{E}_\theta^{EV}(\mathbf{E}_i), \mathbf{EV}_i^{seq}) \in \mathbb{R}^{|\mathbf{EV}_i^{seq}| \times d_{ev}}$. The pointer attention scores, computed as:

$$\text{Pointer}(\mathbf{E}_i, \mathbf{EV}_i^{seq}) = \mathcal{D}_\theta^{EV}(\mathcal{E}_\theta^{EV}(\mathbf{E}_i), \mathbf{EV}_i^{seq}) \cdot \mathcal{E}_\theta^{EV}(\mathbf{E}_i)^T, \quad (7)$$

represent the probability distribution over all possible edges at each position after softmax normalization. Tab. 2 summarizes the output dimensions of each module.

A.4. Details of geometry generation network

Topology-aware Geometric Generation. For the face bounding box generation, we introduce a learnable matrix in $\mathbb{R}^{(M_e+1) \times d_{geom}}$ to encode shared-edge relationships between faces, where d_{geom} denotes the embedding dimension. The attention weights between faces are adjusted by adding the corresponding edge embeddings based on the number of shared edges. In vertex coordinate generation, we incorporate edge connectivity information through two learnable embeddings that indicate whether vertices are connected. These embeddings modulate the attention weights in the Transformer. Additionally, we augment each vertex’s input representation by incorporating the average embedding of its adjacent faces’ bounding boxes. For edge and face geometry generation, unlike the previous stages, we maintain the original attention mechanism without additional adjustment. Instead, we enhance the input token embeddings with topological information. Specifically, for edge geometry generation, we enrich the input token embeddings with: (1) Bounding box embeddings of faces containing the edge. (2) Coordinate embeddings of the edge’s two endpoint vertices. Similarly, for face geometry generation, the input token embeddings are enhanced with: (1) The corresponding face’s bounding box embedding. (2) Coordinate embeddings of the face’s vertices. (3) Geometric embeddings of the face’s boundary edges. This hierarchical

	Input	Embedding	(Masked) Transformer blocks	Output layer
Encoder	$\frac{M_f(M_f-1)}{2}$	$\frac{M_f(M_f-1)}{2} \times d_{ef}$	$\frac{M_f(M_f-1)}{2} \times d_{ef}$	d_{ef}
Decoder	$\frac{M_f(M_f-1)}{2}, d_{ef}$	$\frac{M_f(M_f-1)}{2} \times d_{ef}, d_{ef}$	$\frac{M_f(M_f-1)}{2} \times d_{ef}$	$\frac{M_f(M_f-1)}{2} \times (M_e + 1)$

Table 1. Summary of output shapes for each module of our edge-face model during the training phase.

	Input	Embedding	(Masked) Transformer blocks	Output layer
Encoder	N_e^i	$(2N_e^i + 2) \times d_{ev}$	$(2N_e^i + 2) \times d_{ev}$	-
Decoder	$(2N_e^i + 2) \times d_{ev}, \mathbf{EV}_i^{seq} $	$ \mathbf{EV}_i^{seq} \times d_{ev}$	$ \mathbf{EV}_i^{seq} \times d_{ev}$	$ \mathbf{EV}_i^{seq} \times (2N_e^i + 2)$

Table 2. Summary of output shapes for each module of our edge-vertex model during the training phase.

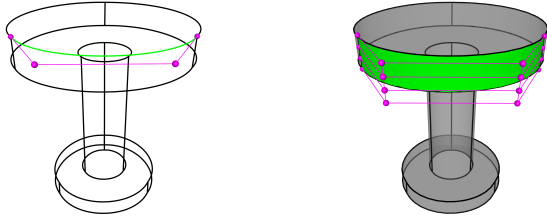


Figure 4. Visualization of B-spline representations in our framework. Each edge is represented by a cubic B-spline with 4 control points, while each face is parameterized by a bi-cubic B-spline surface with a 4×4 control point grid.

approach ensures that each generation stage benefits from both the topological structure and previously generated geometric information, leading to more coherent and physically valid results.

Learning of B-spline Representation. We adopt B-spline representations for both edge and face geometries. For edges, we employ cubic B-splines with knot vector $\{0, 0, 0, 0, 1, 1, 1, 1\}$, while faces are represented using bi-cubic B-splines with knot vectors $\{0, 0, 0, 0, 1, 1, 1, 1\}$ in both u and v directions. This B-spline representation exhibits inherent symmetry, meaning different orderings of the same control points can produce identical geometric forms. To ensure consistency in our training data, we utilize OpenCascade’s curve and surface conversion functions [6] to standardize the B-spline representations. Specifically, all edges and faces in the dataset are converted to B-spline form, establishing a consistent ordering of control points that is maintained throughout the training process. During inference, we generate the control points sequentially using our trained model and construct the B-spline geometries through OpenCascade’s built-in functions, following the same control point ordering convention established during training. Fig. 4 illustrates examples of edge and face geometries in B-spline representation along with their corresponding control points.

B. Experiments

B.1. Implementation Details

We implement our framework in PyTorch and conduct all experiments on 4 NVIDIA A800 GPUs. The B-rep dataset is randomly split into training (90%), validation (5%), and test (5%) sets. The transformer embedding dimensions are set to $d_{ef} = 128$ for the edge-face model, $d_{ev} = 256$ for the edge-vertex model, and $d_{geom} = 512$ for the geometry generation models. To accommodate the topology structure, we set the maximum number of shared edges between faces to $M_e = 5$ and the maximum number of faces to $M_f = 30$. For topology learning, we train the edge-face model for 2,000 epochs and the edge-vertex model for 1,000 epochs. The four geometry generation models are each trained for 3,000 epochs. All models are optimized using Adam with an initial learning rate of 5×10^{-4} (1×10^{-4} for the ABC dataset) and weight decay of 1×10^{-6} , with a batch size of 512. For the geometry generation models, we employ a linear-schedule DDPM [2] with 1,000 diffusion steps during training. The noise schedule follows a linear beta schedule from 1×10^{-4} to 2×10^{-2} .

B.2. Shape Retrieval

To further evaluate DTGBrepGen’s capability in generating novel shapes rather than merely memorizing training samples, we conduct a shape retrieval experiment [3] between the generated samples and the training dataset. Following established protocols, we compute both Chamfer Distance (CD) and Light Field Distance (LFD) [1] between 500 randomly generated shapes and the entire training set. As shown in Fig. 5, we visualize representative examples of our generated shapes alongside their two most similar counterparts from the training set retrieved using both metrics. The distinct geometric variations between generated samples and their nearest neighbors in the training set demonstrate that DTGBrepGen is not simply reproducing training examples. These results collectively validate that our topology-geometry decoupled approach enables the cre-

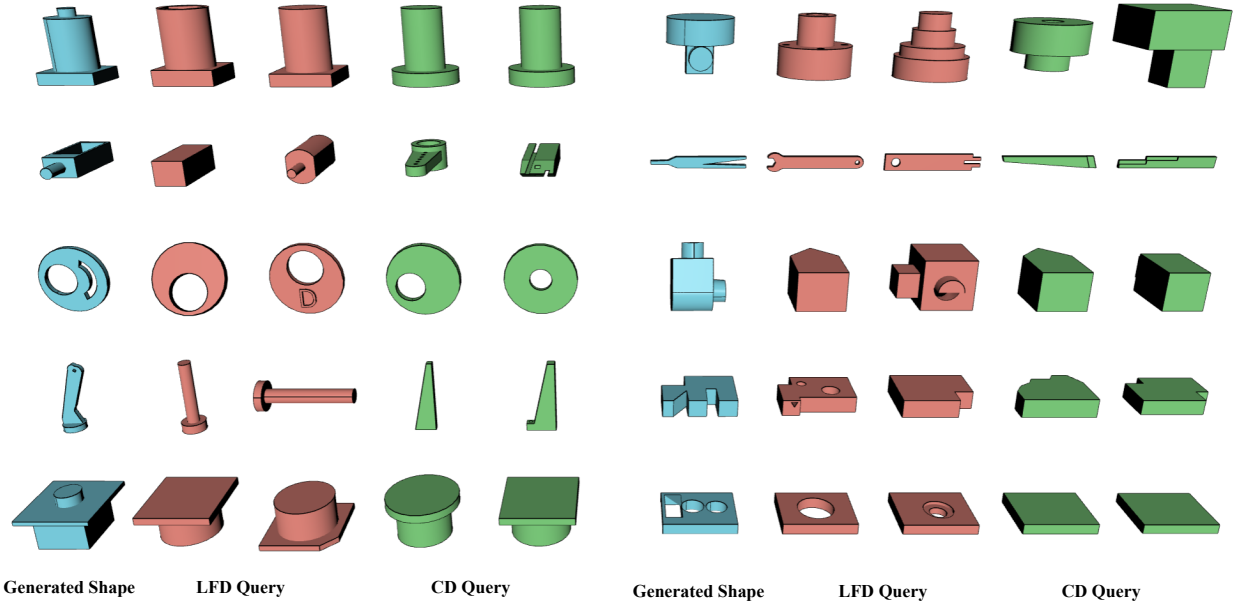


Figure 5. Representative examples of generated shapes alongside their two nearest neighbors from the training set, retrieved using Chamfer Distance and Light Field Distance metrics. The distinct geometric differences highlight the novel nature of the generated shapes.

ation of novel, yet realistic B-rep models that extend beyond the training distribution while maintaining high geometric quality.

B.3. Qualitative Results of the Ablation Study

To demonstrate the advantages of B-spline representation over discrete point-based approaches, we conduct extensive qualitative comparisons on the ABC dataset, as illustrated in Fig. 6. Our B-spline-based method exhibits consistently superior performance, producing smoother and more geometrically accurate details compared to the point-based variant. These qualitative results, aligned with our quantitative findings in the main paper, further validate the effectiveness of directly learning B-spline control point distributions over discrete point-based representations. The superior performance can be attributed to the B-spline representation’s inherent ability to capture continuous geometric features with fewer parameters, leading to more robust and accurate shape generation.

B.4. More Examples Generated by DTGBrepGen

We present additional generation results to demonstrate DTGBrepGen’s versatility. Fig. 7 shows diverse unconditional generation examples on the DeepCAD dataset. Fig. 8 presents class-conditioned generation results across different furniture categories. Fig. 9 illustrates our model’s capability in translating point clouds to B-rep models.

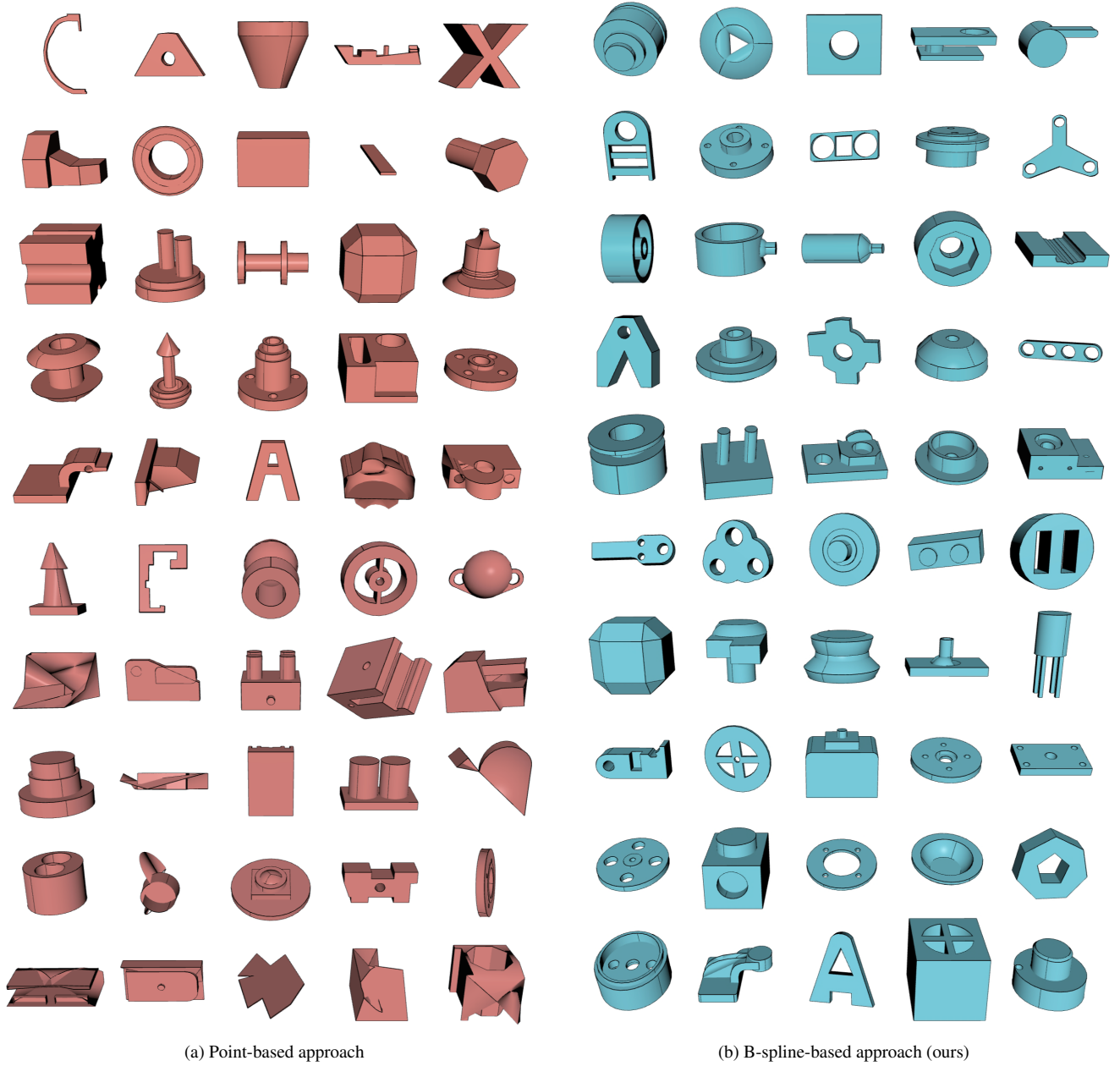


Figure 6. Qualitative comparison between B-spline-based and point-based geometric representations on the ABC dataset.

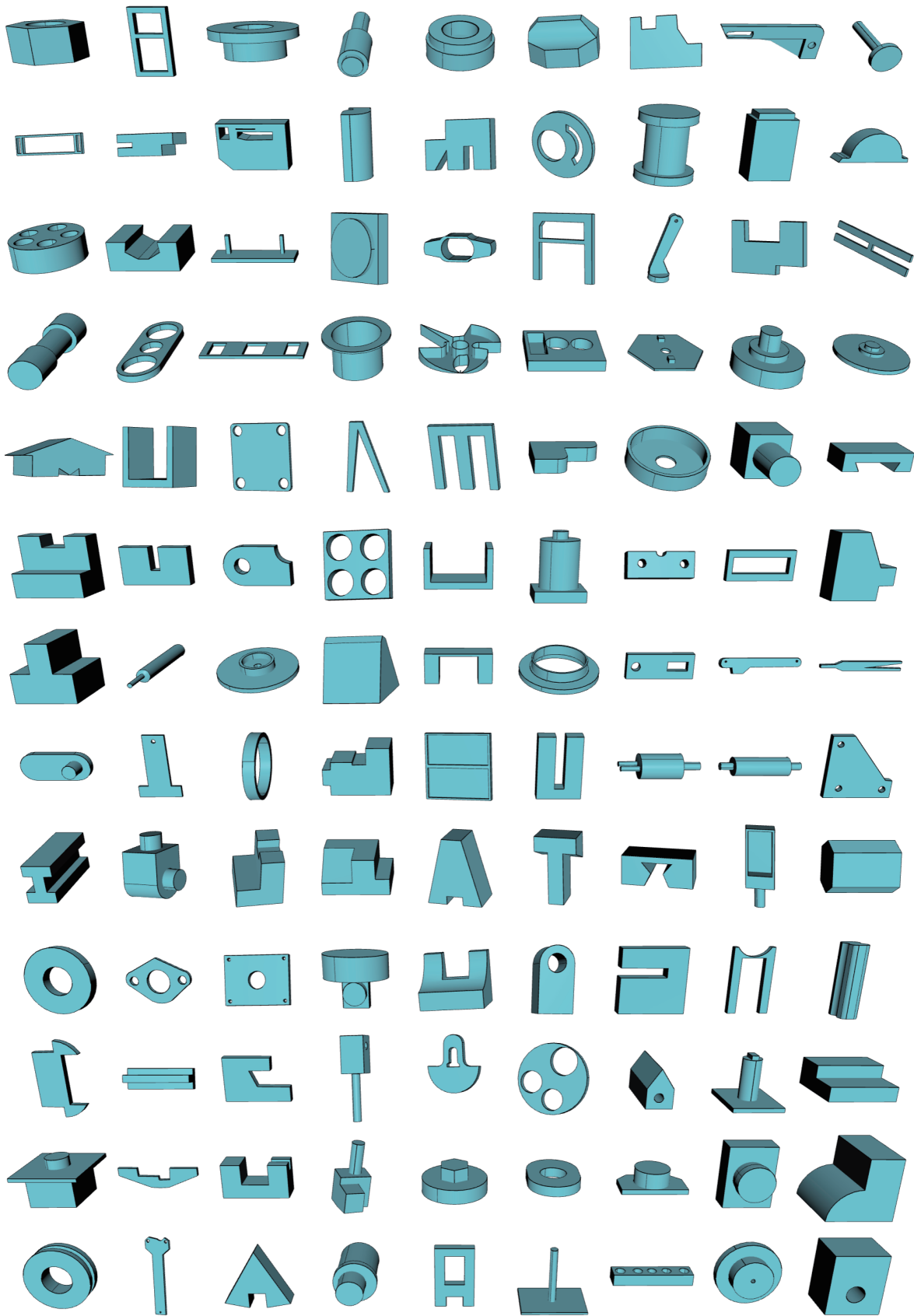


Figure 7. Examples of B-rep models generated by DTGBrepGen on the DeepCAD dataset.



Figure 8. Examples of class-conditioned generation on the Furniture dataset, with distinct colors representing different categories.

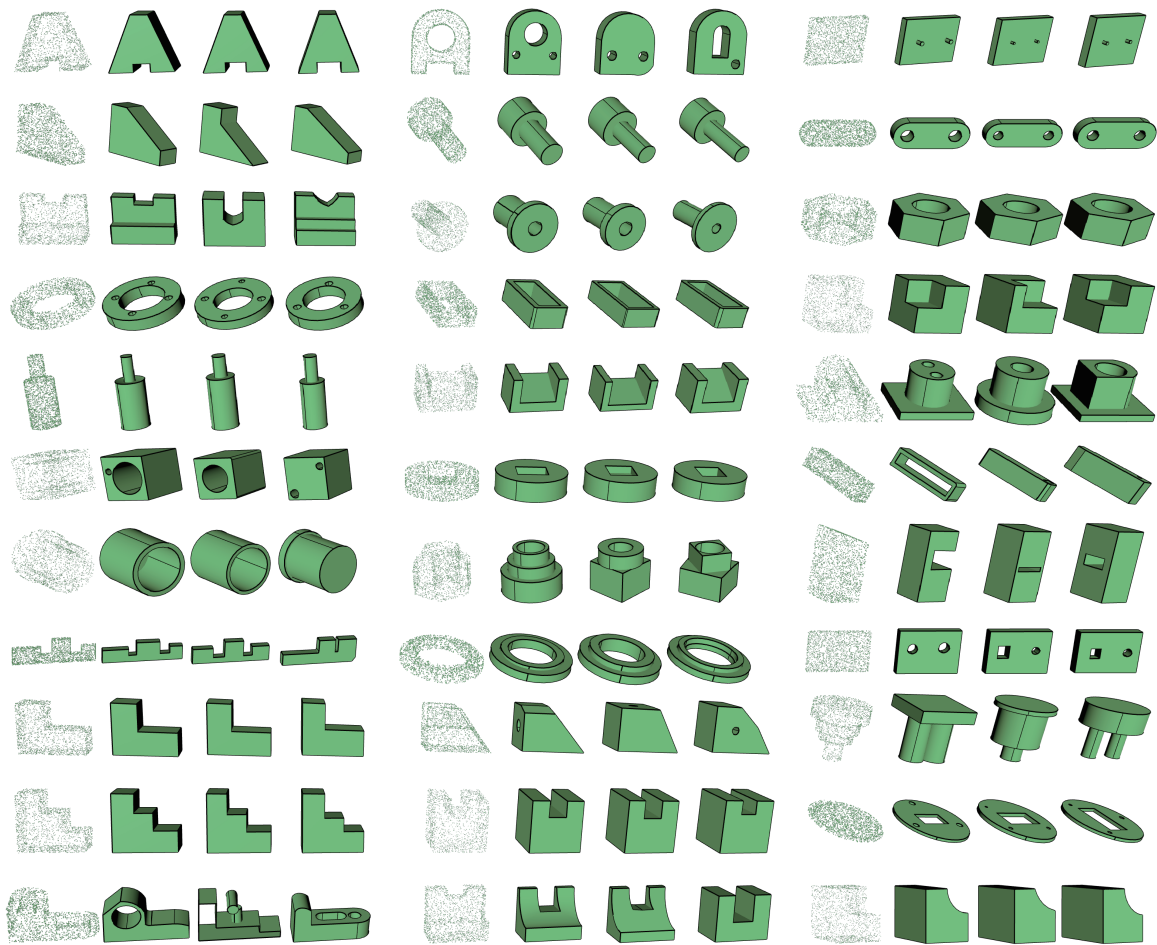


Figure 9. Examples of point cloud-conditioned B-rep generation on the DeepCAD dataset.

References

- [1] Ding-Yun Chen, Xiao-Pei Tian, Yu-Te Shen, and Ming Ouhyoung. On visual similarity based 3d model retrieval. In *Computer Graphics Forum*, pages 223–232. Wiley Online Library, 2003. 4
- [2] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 33:6840–6851, 2020. 4
- [3] Ka-Hei Hui, Ruihui Li, Jingyu Hu, and Chi-Wing Fu. Neural wavelet-domain diffusion for 3d shape generation. 2022. 4
- [4] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014. 2
- [5] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017. 3
- [6] Trevor Laughlin. pyocct – python bindings for opencascade via pybind11, 2020. <https://github.com/trelau/pyOCCT>. 4
- [7] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017. 2