DeepCAD: A Deep Generative Network for Computer-Aided Design Models

Rundi Wu   Chang Xiao   Changxi Zheng
Columbia University

`{rundi, chang, cxz}@cs.columbia.edu`

Abstract

Deep generative models of 3D shapes have received a great deal of research interest. Yet, almost all of them generate discrete shape representations, such as voxels, point clouds, and polygon meshes. We present the first 3D generative model for a drastically different shape representation— describing a shape as a sequence of computer-aided design (CAD) operations. Unlike meshes and point clouds, CAD models encode the user creation process of 3D shapes, widely used in numerous industrial and engineering design tasks. However, the sequential and irregular structure of CAD operations poses significant challenges for existing 3D generative models. Drawing an analogy between CAD operations and natural language, we propose a CAD generative network based on the Transformer. We demonstrate the performance of our model for both shape autoencoding and random shape generation. To train our network, we create a new CAD dataset consisting of 178,238 models and their CAD construction sequences. We have made this dataset publicly available to promote future research on this topic.

1. Introduction

It is our human nature to imagine and invent, and to express our invention in 3D shapes. This is what the paper and pencil were used for when Leonardo da Vinci sketched his mechanisms; this is why such drawing tools as the parallel bar, the French curve, and the divider were devised; and this is wherefore, in today's digital era, the computer aided design (CAD) software have been used for 3D shape creation in a myriad of industrial sectors, ranging from automotive and aerospace to manufacturing and architectural design.

Can the machine also invent 3D shapes? Leveraging the striking advance in generative models of deep learning, lots of recent research efforts have been directed to the generation of 3D models. However, existing 3D generative models merely create computer discretization of 3D shapes: 3D point clouds [6, 52, 53, 8, 30], polygon meshes [17, 42, 31], and levelset fields [12, 33, 29, 50, 11]. Still missing is the ability to generate the very nature of 3D shape design—the drawing process.
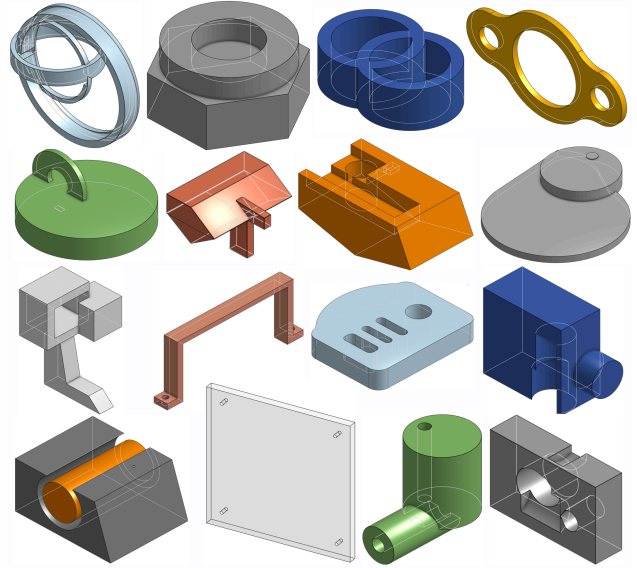

Figure 1. **A gallery of generated CAD designs.** Our generative network is able to produce a diverse range of CAD designs. Each CAD model consists of a sequence of CAD operations with specific parameters. The resulting 3D shapes are clean, have sharp geometric features, and can be readily user-edited.

We propose a deep generative network that outputs a sequence of operations used in CAD tools (such as SolidWorks and AutoCAD) to construct a 3D shape. Generally referred as a *CAD model*, such an operational sequence represents the "drawing" process of shape creation. Today, almost all the industrial 3D designs start with CAD models. Only until later in the production pipeline, if needed, they are discretized into polygon meshes or point clouds.

To our knowledge, this is the first work toward a generative model of CAD designs. The challenge lies in the CAD design's sequential and parametric nature. A CAD model consists of a series of geometric operations (e.g., curve sketch, extrusion, fillet, boolean, chamfer), each controlled by certain parameters. Some of the parameters are discrete options; others have continuous values (more discussion in Sec. 3.1). These irregularities emerge from the user creation process of 3D shapes, and thus contrast starkly to the discrete 3D representations (i.e., voxels, point clouds,

and meshes) used in existing generative models. In consequence, previously developed 3D generative models are unsuited for CAD model generation.

Technical contributions. To overcome these challenges, we seek a representation that reconciles the irregularities in CAD models. We consider the most frequently used CAD operations (or commands), and unify them in a common structure that encodes their command types, parameters, and sequential orders. Next, drawing an analogy between CAD command sequences and natural languages, we propose an autoencoder based on the Transformer network [40]. It embeds CAD models into a latent space, and later decode a latent vector into a CAD command sequence. To train our autoencoder, we further create a new dataset of CAD command sequences, one that is orders of magnitude larger than the existing dataset of the same type. We have also made this dataset publicly available[1] to promote future research on learning-based CAD designs.

Our method is able to generate plausible and diverse CAD designs (see Fig. 1). We carefully evaluate its generation quality through a series of ablation studies. Lastly, we end our presentation with an outlook on useful applications enabled by our CAD autoencoder.

2. Related work

Parametric shape inference. Advance in deep learning has enabled neural network models that analyze geometric data and infer parametric shapes. ParSeNet [38] decomposes a 3D point cloud into a set of parametric surface patches. PIE-NET [43] extracts parametric boundary curves from 3D point clouds. UV-Net [19] and BrepNet [24] focus on encoding a parametric model's boundary curves and surfaces. Li et al. [25] trained a neural network on synthetic data to convert 2D user sketches into CAD operations. Recently, Xu et al. [51] applied neural-guided search to infer CAD modeling sequence from parametric solid shapes.

Generative models of 3D shapes. Recent years have also witnessed increasing research interests on deep generative models for 3D shapes. Most existing methods generate 3D shapes in *discrete* forms, such as voxelized shapes [49, 16, 27, 26], point clouds [6, 52, 53, 8, 30], polygon meshes [17, 42, 31], and implicit signed distance fields [12, 33, 29, 50, 11]. The resulting shapes may still suffer from noise, lack sharp geometric features, and are not directly user editable.

Therefore, more recent works have sought neural network models that generate 3D shape as a series of geometric operations. CSGNet [37] infers a sequence of Constructive Solid Geometry (CSG) operations based on voxelized shape input; and UCSG-Net [21] further advances the inference with no supervision from ground truth CSG trees. Other than using CSG operations, several works synthesize

3D shapes using their proposed domain specific languages (DSLs) [39, 41, 30, 20]. For example, Jones et al. [20] proposed ShapeAssembly, a DSL that constructs 3D shapes by structuring cuboid proxies in a hierarchical and symmetrical fashion, and this structure can be generated through a variational autoencoder.

In contrast to all these works, our autoencoder network outputs CAD models specified as a sequence of CAD operations. CAD models have become the standard shape representation in almost every sectors of industrial production. Thus, the output from our network can be readily imported into any CAD tools [1, 2, 3] for user editing. It can also be directly converted into other shape formats such as point clouds and polygon meshes. To our knowledge, this is the first generative model directly producing CAD designs.

Transformer-based models. Technically, our work is related to the Transformer network [40], which was introduced as an attention-based building block for many natural language processing tasks [13]. The success of the Transformer network has also inspired its use in image processing tasks [34, 9, 14] and for other types of data [31, 10, 44]. Concurrent works [47, 32, 15] on constrained CAD sketches generation also rely on Transformer network.

Also related to our work is DeepSVG [10], a Transformer-based network for the generation of Scalable Vector Graphic (SVG) images. SVG images are described by a collection of parametric primitives (such as lines and curves). Apart from limited in 2D, those primitives are grouped with no specific order or dependence. In contrast, CAD commands are described in 3D; they can be interdependent (e.g., through CSG boolean operations) and must follow a specific order. We therefore seek a new way to encode CAD commands and their sequential order in a Transformer-based autoencoder.

3. Method

We now present our DeepCAD model, which revolves around a new representation of CAD command sequences (Sec. 3.1.2). Our CAD representation is specifically tailored, for feeding into neural networks such as the proposed Transformer-based autoencoder (Sec. 3.2). It also leads to a natural objective function for training (Sec. 3.4). To train our network, we create a new dataset, one that is significantly larger than existing datasets of the same type (Sec. 3.3), and one that itself can serve beyond this work for future research.

3.1. CAD Representation for Neural Networks

The CAD model offers two levels of representation. At the user-interaction level, a CAD model is described as a sequence of operations that the user performs (in CAD software) to create a solid shape—for example, a user may `sketch` a closed curve profile on a 2D plane, and then `extrude` it into a 3D solid shape, which is further processed by other operations such as a boolean `union` with another

---

[1] Code and data are available here.

CAD construction process:

Parametrized command sequence:

$\langle\mathtt{SOL}\rangle_1 : \emptyset$ $\mathtt{E}_8 : (0, 0, 0, -2, -1, 0, 3,$
    $\mathtt{L}_2 : (2, 0)$ $1, 0, \text{New body}, \text{One-sided})$
    $\mathtt{A}_3 : (2, 2, \pi, 1)$ $\langle\mathtt{SOL}\rangle_9 : \emptyset$
    $\mathtt{L}_4 : (0, 2)$ $\mathtt{R}_{10} : (0, 0, 1.125)$
    $\mathtt{L}_5 : (0, 0)$ $\mathtt{E}_{11} : (0, 0, 0, -2, 0, 0, 2.25,$
$\langle\mathtt{SOL}\rangle_6 : \emptyset$ $2, 0, \text{Join}, \text{One-sided})$
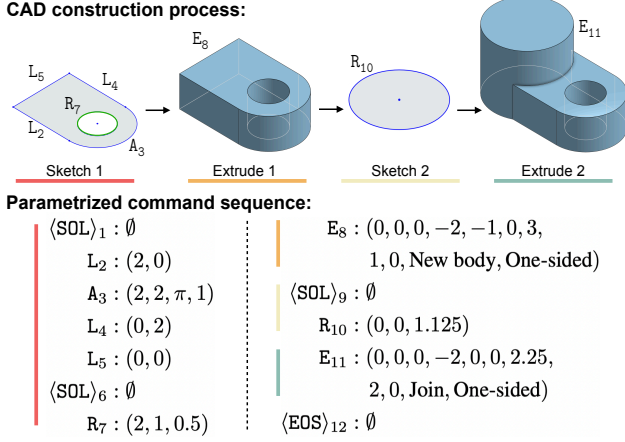 $\mathtt{R}_7 : (2, 1, 0.5)$ $\langle\mathtt{EOS}\rangle_{12} : \emptyset$

Figure 2. **A CAD model example** specified by the commands in Table 1. (Top) the CAD model's construction sequence, annotated with the command types. (Bottom) the command sequence description of the model. Parameter normalization and quantization are not shown in this case. In "Sketch 1", $\mathtt{L}_2$-$\mathtt{A}_3$-$\mathtt{L}_4$-$\mathtt{L}_5$ forms a loop (in blue) and $\mathtt{C}_7$ forms another loop (in green), and the two loops bounds a sketch profile (in gray).

already created solid shape (see Fig. 2). We refer to such a specification as a CAD *command sequence*.

Behind the command sequence is the CAD model's kernel representation, widely known as the *boundary representation* (or B-rep) [45, 46]. Provided a command sequence, its B-rep is automatically computed (often through the industry standard library Parasolid). It consists of topological components (i.e., vertices, parametric edges and faces) and the connections between them to form a solid shape.

In this work, we aim for a generative model of CAD command sequences, not B-reps. This is because the B-rep is an abstraction from the command sequence: a command sequence can be easily converted into a B-rep, but the converse is hard, as different command sequences may result in the same B-rep. Moreover, a command sequence is human-interpretable; it can be readily edited (e.g., by importing them into CAD tools such as AutoCAD and Onshape), allowing them to be used in various downstream applications.

3.1.1 Specification of CAD Commands

Full-fledged CAD tools support a rich set of commands, although in practice only a small fraction of them are commonly used. Here, we consider a subset of the commands that are of frequent use (see Table 1). These commands fall into two categories, namely *sketch* and *extrusion*. While conceptually simple, they are sufficiently expressive to generate a wide variety of shapes, as has been demonstrated in [48].

Sketch. Sketch commands are used to specify closed curves on a 2D plane in 3D space. In CAD terminology, each closed curve is referred as a *loop*, and one or more loops form a closed region called a *profile* (see "Sketch 1" in Fig. 2). In our representation, a profile is described by

Commands	Parameters
$\langle\mathtt{SOL}\rangle$	$\emptyset$
L (Line)	x, y : line end-point
A (Arc)	x, y : arc end-point
	α : sweep angle
	f : counter-clockwise flag
R (Circle)	x, y : center
	r : radius
E (Extrude)	θ, ϕ, γ : sketch plane orientation
	p_x, p_y, p_z : sketch plane origin
	s : scale of associated sketch profile
	e_1, e_2 : extrude distances toward both sides
	b : boolean type, u : extrude type
$\langle\mathtt{EOS}\rangle$	$\emptyset$

Table 1. **CAD commands** and their parameters. $\langle\mathtt{SOL}\rangle$ indicates the start of a loop; $\langle\mathtt{EOS}\rangle$ indicates the end of the whole sequence.

a list of loops on its boundary; a loop always starts with an indicator command $\langle\mathtt{SOL}\rangle$ followed by a series of curve commands C_i. We list all the curves on the loop in counter-clockwise order, beginning with the curve whose starting point is at the most bottom-left; and the loops in a profile are sorted according to the bottom-left corners of their bounding boxes. Figure 2 illustrates two sketch profiles.

In practice, we consider three kinds of curve commands that are the most widely used: draw a line, an arc, and a circle. While other curve commands can be easily added (see Sec. 5), statistics from our large-scale real-world dataset (described in Sec. 3.3) show that these three types of commands constitute 92% of the cases.

Each curve command C_i is described by its curve type $t_i \in \{\langle\mathtt{SOL}\rangle, \mathtt{L}, \mathtt{A}, \mathtt{R}\}$ and its parameters listed in Table 1. Curve parameters specify the curve's 2D location in the sketch plane's local frame of reference, whose own position and orientation in 3D will be described shortly in the associated extrusion command. Since the curves in each loop are concatenated one after another, for the sake of compactness we exclude the curve's starting position from its parameter list; each curve always starts from the ending point of its predecessor in the loop. The first curve always starts from the origin of the sketch plane, and the world-space coordinate of the origin is specified in the extrusion command.

In short, a sketch profile S is described by a list of loops $S = [Q_1, \ldots, Q_N]$, where each loop Q_i consists of a series of curves starting from the indicator command $\langle\mathtt{SOL}\rangle$ (i.e., $Q_i = [\langle\mathtt{SOL}\rangle, C_1, \ldots, C_{n_i}]$), and each curve command $C_j = (t_j, \boldsymbol{p}_j)$ specifies the curve type t_i and its shape parameters $\boldsymbol{p}_j$ (see Fig. 2).

Extrusion. The extrusion command serves two purposes. **1)** It extrudes a sketch profile from a 2D plane into a 3D body, and the extrusion type can be either *one-sided*, *symmetric*, or

two-sided with respect to the profile's sketch plane. **2)** The command also specifies (through the parameter b in Table 1) how to merge the newly extruded 3D body with the previously created shape by one of the boolean operations: either creating a *new* body, or *joining*, *cutting* or *intersecting* with the existing body.

The extruded profile—which consists of one or more curve commands—is always referred to the one described immediately before the extrusion command. The extrusion command therefore needs to define the 3D orientation of that profile's sketch plane and its 2D local frame of reference. This is defined by a rotational matrix, determined by (θ, γ, ϕ) parameters in Table 1. This matrix is to align the world frame of reference to the plane's local frame of reference, and to align z-axis to the plane's normal direction. In addition, the command parameters include a scale factor s of the extruded profile; the rationale behind this scale factor will be discussed in Sec. 3.1.2.

With these commands, we describe a CAD model M as a sequence of curve commands interleaved with extrusion commands (see Fig. 2). In other words, M is a command sequence $M = [C_1, \ldots, C_{N_c}]$, where each C_i has the form $(t_i, \boldsymbol{p}_i)$ specifying the command type t_i and parameters $\boldsymbol{p}_i$.

3.1.2 Network-friendly Representation

Our specification of a CAD model M is akin to natural language. The vocabulary consists of individual CAD commands expressed sequentially to form sentences. The subject of a sentence is the sketch profile; the predicate is the extrusion. This analogy suggests that we may leverage the network structures, such as the Transformer network [40], succeeded in natural language processing to fulfill our goal.

However, the CAD commands also differ from natural language in several aspects. Each command has a different number of parameters. In some commands (e.g., the extrusion), the parameters are a mixture of both continuous and discrete values, and the parameter values span over different ranges (recall Table 1). These traits render the command sequences ill-posed for direct use in neural networks.

To overcome this challenge, we regularize the dimensions of command sequences. First, for each command, its parameters are stacked into a 16×1 vector, whose elements correspond to the collective parameters of all commands in Table 1 (i.e., $\boldsymbol{p}_i = [x, y, \alpha, f, r, \theta, \phi, \gamma, p_x, p_y, p_z, s, e_1, e_2, b, u]$). Unused parameters for each command are simply set to be -1. Next, we fix the total number N_c of commands in every CAD model M. This is done by padding the CAD model's command sequence with the empty command $\langle\texttt{EOS}\rangle$ until the sequence length reaches N_c. In practice, we choose $N_c = 60$, the maximal command sequence length appeared in our training dataset.

Furthermore, we unify continuous and discrete parameters by quantizing the continuous parameters. To this end,
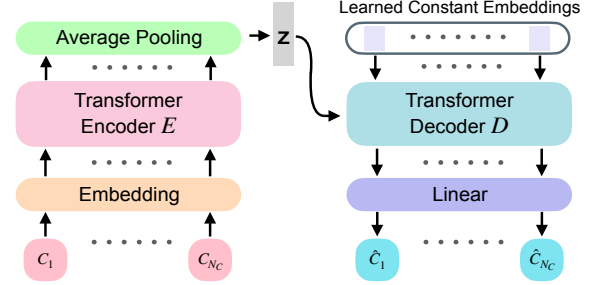

Figure 3. **Our network architecture.** The input CAD model, represented as a command sequence $M = \{C_i\}_{i=1}^{N_c}$ is first projected to an embedding space and then fed to the encoder E resulting in a latent vector $\boldsymbol{z}$. The decoder D takes learned constant embeddings as input, and also attends to the latent vector $\boldsymbol{z}$. It then outputs the predicted command sequence $\hat{M} = \{\hat{C}_i\}_{i=1}^{N_c}$..

we normalize every CAD model within a $2 \times 2 \times 2$ cube; we also normalize every sketch profile within its bounding box, and include a scale factor s (in extrusion command) to restore the normalized profile into its original size. The normalization restricts the ranges of continuous parameters, allowing us to quantize their values into 256 levels and express them using 8-bit integers. As a result, all the command parameters possess only discrete sets of values.

Not simply is the parameter quantization a follow-up of the common practice for training Transformer-based networks [36, 31, 44]. Particularly for CAD models, it is crucial for improving the generation quality (as we empirically confirm in Sec. 4.1). In CAD designs, certain geometric relations—such as parallel and perpendicular sketch lines—must be respected. However, if a generative model directly generates continuous parameters, their values, obtained through parameter regression, are prone to errors that will break these strict relations. Instead, parameter quantization allows the network to "classify" parameters into specific levels, and thereby better respect learned geometric relations.

In Sec. 4.1, we will present ablation studies that empirically justify our choices of CAD command representation.

3.2. Autoencoder for CAD Models

We now introduce an autoencoder network that leverages our representation of CAD commands. Figure 3 illustrates its structure, and more details are provided in Sec. C of supplementary document. Once trained, the decoder part of the network will serve naturally as a CAD generative model.

Our autoencoder is based on the Transformer network, inspired by its success for processing sequential data [40, 13, 28]. Our autoencoder takes as input a CAD command sequence $M = [C_1, \cdots, C_{N_c}]$, where N_c is a fixed number (recall Sec. 3.1.2). First, each command C_i is projected separately onto a continuous embedding space of dimension $d_E = 256$. Then, all the embeddings are put together to feed into an encoder E, which in turn outputs a latent vector $\boldsymbol{z} \in \mathbb{R}^{256}$. The decoder takes the latent vector $\boldsymbol{z}$ as input,

and outputs a generated CAD command sequence $\hat{M}$.

Embedding. Similar in spirit to the approach in natural language processing [40], we first project every command C_i onto a common embedding space. Yet, different from words in natural languages, a CAD command $C_i = (t_i, \boldsymbol{p}_i)$ has two distinct parts: its command type t_i and parameters $\boldsymbol{p}_i$. We therefore formulate a different way of computing the embedding of C_i: take it as a sum of three embeddings, that is, $\boldsymbol{e}(C_i) = \boldsymbol{e}_i^{\text{cmd}} + \boldsymbol{e}_i^{\text{param}} + \boldsymbol{e}_i^{\text{pos}} \in \mathbb{R}^{d_{\text{E}}}$.

The first embedding $\boldsymbol{e}_i^{\text{cmd}}$ accounts for the command type t_i, given by $\boldsymbol{e}_i^{\text{cmd}} = \mathsf{W}_{\text{cmd}}\boldsymbol{\delta}_i^{\text{c}}$. Here $\mathsf{W}_{\text{cmd}} \in \mathbb{R}^{d_{\text{E}} \times 6}$ is a learnable matrix and $\boldsymbol{\delta}_i^{\text{c}} \in \mathbb{R}^6$ is a one-hot vector indicating the command type t_i among the six command types.

The second embedding $\boldsymbol{e}_i^{\text{param}}$ considers the command parameters. As introduced in Sec. 3.1.2, every command has 16 parameters, each of which is quantized into an 8-bit integer. We convert each of these integers into a one-hot vector $\boldsymbol{\delta}_{i,j}^{\text{p}}$ ($j = 1..16$) of dimension $2^8 + 1 = 257$; the additional dimension is to indicate that the parameter is unused in that command. Stacking all the one-hot vectors into a matrix $\boldsymbol{\delta}_i^{\text{p}} \in \mathbb{R}^{257 \times 16}$, we embed each parameter separately using another learnable matrix $\mathsf{W}_{\text{param}}^b \in \mathbb{R}^{d_{\text{E}} \times 257}$, and then combine the individual embeddings through a linear layer $\mathsf{W}_{\text{param}}^a \in \mathbb{R}^{d_{\text{E}} \times 16 d_{\text{E}}}$, namely,

$$\boldsymbol{e}_i^{\text{param}} = \mathsf{W}_{\text{param}}^a \text{flat}(\mathsf{W}_{\text{param}}^b \boldsymbol{\delta}_i^{\text{p}}), \qquad (1)$$

where flat$(\cdot)$ flattens the input matrix to a vector.

Lastly, similar to [40], the positional embedding $\boldsymbol{e}_i^{\text{pos}}$ is to indicate the index of the command C_i in the whole command sequence, defined as $\boldsymbol{e}_i^{\text{pos}} = \mathsf{W}_{\text{pos}}\boldsymbol{\delta}_i$, where $\mathsf{W}_{\text{pos}} \in \mathbb{R}^{d_{\text{E}} \times N_c}$ is a learnable matrix and $\boldsymbol{\delta}_i \in \mathbb{R}^{N_c}$ is the one-hot vector filled with 1 at index i and 0 otherwise.

Encoder. Our encoder E is composed of four layers of Transformer blocks, each with eight attention heads and feed-forward dimension of 512. The encoder takes the embedding sequence $[\boldsymbol{e}_1, .., \boldsymbol{e}_{N_c}]$ as input, and outputs vectors $[\boldsymbol{e}_1', .., \boldsymbol{e}_{N_c}']$; each has the same dimension $d_{\text{E}} = 256$. The output vectors are finally averaged to produce a single d_{E}-dimensional latent vector $\boldsymbol{z}$.

Decoder. Also built on Transformer blocks, our decoder D has the same hyper-parameter settings as the encoder. It takes as input learned constant embeddings while also attending to the latent vector $\boldsymbol{z}$—similar input structure has been used in [9, 10]. Output from the last Transformer block is fed into a linear layer to predict a CAD command sequence $\hat{M} = [\hat{C}_1, .., \hat{C}_{N_c}]$, including both the command type $\hat{t}_i$ and parameters $\hat{\boldsymbol{p}}_i$ for each command. As opposed to the autoregressive strategy commonly used in natural language processing [40], we adopt the feed-forward strategy [9, 10], and the prediction of our model can be factorized as

$$p(\hat{M}|z, \Theta) = \prod_{i=1}^{N_c} p(\hat{t}_i, \hat{\boldsymbol{p}}_i | z, \Theta), \qquad (2)$$

where Θ denotes network parameters of the decoder.

3.3. Creation of CAD Dataset

Several datasets of CAD designs exist, but none of them suffice for our training. In particular, the ABC dataset [23] collects about 1 million CAD designs from Onshape, a web-based CAD tool and repository [3]. Although this is a large-scale dataset, its CAD designs are provided in B-rep format, with no sufficient information to recover how the designs are constructed by CAD operations. The recent *Fusion 360 Gallery* dataset [48] offers CAD designs constructed by profile sketches and extrusions, and it provides the CAD command sequence for each design. However, this dataset has only ~ 8000 CAD designs, not enough for training a well generalized generative model.

We therefore create a new dataset that is large-scale and provides CAD command sequences. Apart from using it to train our autoencoder network, this dataset may also serve for future research. We have made it publicly available.

To create the dataset, we also leverage Onshape's CAD repository and its developer API [4] to parse the CAD designs. We start from the ABC dataset. For each CAD model, the dataset provides a link to Onshape's original CAD design. We then use Onshape's domain specific language (called FeatureScript [5]) to parse CAD operations and parameters used in that design. For CAD models that use the operations beyond sketch and extrusion, we simply discard them. For the rest of the models, we use a FeatureScript program to extract the sketch profiles and extrusions, and express them using the commands listed in Table 1.

In the end, we collect a dataset with 178,238 CAD designs all described as CAD command sequences. This is orders of magnitude larger than the existing dataset of the same type [48]. The dataset is further split into training, validation and test sets by 90%-5%-5% in a random fashion, ready to use in training and testing. Figure 9 in the supplementary document samples some CAD models from our dataset.

3.4. Training and Runtime Generation

Training. Leveraging the dataset, we train our autoencoder network using the standard Cross-Entropy loss. Formally, we define the loss between the predicted CAD model $\hat{M}$ and the ground truth model M as

$$\mathcal{L} = \sum_{i=1}^{N_c} \ell(\hat{t}_i, t_i) + \beta \sum_{i=1}^{N_c} \sum_{j=1}^{N_P} \ell(\hat{\boldsymbol{p}}_{i,j}, \boldsymbol{p}_{i,j}), \qquad (3)$$

where $\ell(\cdot, \cdot)$ denotes the standard Cross-Entropy, N_p is the number of parameters ($N_p = 16$ in our examples), and β is a weight to balance both terms ($\beta = 2$ in our examples). Note that in the ground-truth command sequence, some commands are empty (i.e., the padding command $\langle \text{EOS} \rangle$) and some command parameters are unused (i.e., labeled as

Method	ACC$_{cmd}$ ↑	ACC$_{param}$ ↑	median CD↓	Invalid Ratio↓
Ours+Aug	**99.50**	**97.98**	**0.752**	**2.72**
Ours	99.36	97.47	0.787	3.30
Alt-ArcMid	99.34	97.31	0.790	3.26
Alt-Trans	99.33	97.56	0.792	3.30
Alt-Rel	99.33	97.66	0.863	3.51
Alt-Regr	-	-	2.142	4.32

Table 2. **Quantitative evaluation of autoencoding.** ACC$_{cmd}$ and ACC$_{param}$ are both multiplied by 100%, and CD is multiplied by 10^3. ↑: a higher metric value indicates better autoencoding quality. ↓: a lower metric value is better. ACC values for `Alt-Regr` are not available since `Alt-Regr` does not use quantized parameters.

-1). In those cases, their corresponding contributions to the summation terms in (3) are simply ignored.

The training process uses the Adam optimizer [22] with a learning rate 0.001 and a linear warm-up period of 2000 initial steps. We set a dropout rate of 0.1 for all Transformer blocks and apply gradient clipping of 1.0 in backpropagation. We train the network for 1000 epochs with a batch size of 512.

CAD generation. Once the autoencoder is well trained, we can represent a CAD model using a 256-dimensional latent vector z. For automatic generation of CAD models, we employ the latent-GAN technique [6, 12, 50] on our learned latent space. The generator and discriminator are both as simple as a multilayer perceptron (MLP) network with four hidden layers, and they are trained using Wasserstein-GAN training strategy with gradient penalty [7, 18]. In the end, to generate a CAD model, we sample a random vector from a multivariate Gaussian distribution and feeding it into the GAN's generator. The output of the GAN is a latent vector z input to our Transformer-based decoder.

4. Experiments

In this section, we evaluate our autoencoder network from two perspectives: the autoencoding of CAD models (Sec. 4.1) and latent-space shape generation (Sec. 4.2). We also discuss possible applications that can benefit from our CAD generative model (Sec. 4.3).

There exist no previous generative models for CAD designs, and thus no methods for our model to direct compare with. Our goal here is to understand the performance of our model under different metrics, and justify the algorithmic choices in our model through a series of ablation studies.

4.1. Autoencoding of CAD Models

The autoencoding performance has often been used to indicate the extent to which the generative model can express the target data distribution [6, 12, 17]. Here we use our autoencoder network to encode a CAD model M absent
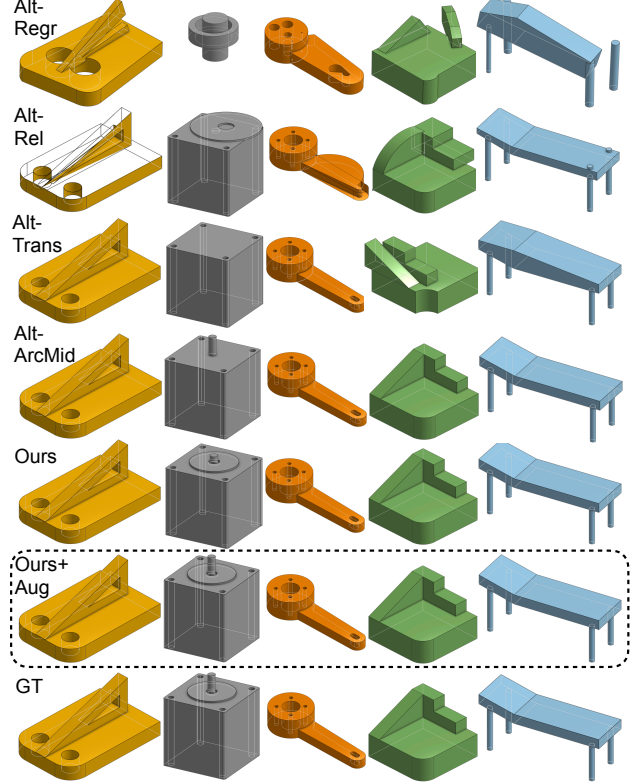

Figure 4. **Comparison of autoencoding results.** Hidden edges are also rendered visible (white). Ground truth (GT) is shown in the bottom row. Our best results are highlighted in the dash-line box.

from the training dataset; we then decode the resulting latent vector into a CAD model $\hat{M}$. The autoencoder is evaluated by the difference between M and $\hat{M}$.

Metrics. To thoroughly understand our autoencoder's performance, we measure the difference between M and $\hat{M}$ in terms of both the CAD commands and the resulting 3D geometry. We propose to evaluate command accuracy using two metrics, namely *Command Accuracy* (ACC$_{cmd}$) and *Parameter Accuracy* (ACC$_{param}$). The former measures the correctness of the predicted CAD command type, defined as

$$\text{ACC}_{cmd} = \frac{1}{N_c} \sum_{i=1}^{N_c} \mathbb{I}[t_i = \hat{t}_i]. \tag{4}$$

Here the notation follows those in Sec. 3. N_c denote the total number of CAD commands, and t_i and $\hat{t}_i$ are the ground-truth and recovered command types, respectively. $\mathbb{I}[\cdot]$ is the indicator function (0 or 1).

Once the command type is correctly recovered, we also evaluate the correctness of the command parameters. This is what Parameter Accuracy (ACC$_{param}$) is meant to measure:

$$\text{ACC}_{param} = \frac{1}{K} \sum_{i=1}^{N_c} \sum_{j=1}^{|\hat{p}_i|} \mathbb{I}[|\boldsymbol{p}_{i,j} - \hat{\boldsymbol{p}}_{i,j}| < \eta] \mathbb{I}[t_i = \hat{t}_i], \tag{5}$$

where $K = \sum_{i=1}^{N_c} \mathbb{I}[t_i = \hat{t}_i]|\boldsymbol{p}_i|$ is the total number of parameters in all correctly recovered commands. Note that $\boldsymbol{p}_{i,j}$ and $\hat{\boldsymbol{p}}_{i,j}$ are both quantized into 8-bit integers. η is chosen as a tolerance threshold accounting for the parameter quantization. In practice, we use $\eta = 3$ (out of 256 levels).

To measure the quality of recovered 3D geometry, we use *Chamfer Distance* (CD), the metric used in many previous generative models of discretized shapes (such as point clouds) [6, 17, 12]. Here, we evaluate CD by uniformly sampling 2000 points on the surfaces of reference shape and recovered shape, respectively; and measure CD between the two sets of points. Moreover, it is not guaranteed that the output CAD command sequence always produces a valid 3D shape. In rare cases, the output commands may lead to an invalid topology, and thus no point cloud can be extracted from that CAD model. We therefore also report the *Invalid Ratio*, the percentage of the output CAD models that fail to be converted to point clouds.

Comparison methods. Due to the lack of existing CAD generative models, we compare our model with several variants in order to justify our data representation and training strategy. In particular, we consider the following variants.

`Alt-Rel` represents curve positions relative to the position of its predecessor curve in the loop. It contrasts to our model, which uses absolute positions in curve specification.

`Alt-Trans` includes in the extrusion command the starting point position of the loop (in addition to the origin of the sketch plane). Here the starting point position and the plane's origin are in the world frame of reference of the CAD model. In contrast, our proposed method includes only the sketch plane's origin, and the origin is translated to the loop's starting position—it is therefore more compact.

`Alt-ArcMid` specifies an arc using its ending and middle point positions, but not the sweeping angle and the counterclockwise flag used in Table 1.

`Alt-Regr` regresses all parameters of the CAD commands using the standard mean-squared error in the loss function. Unlike the model we propose, there is no need to quantize continuous parameters in this approach.

`Ours+Aug` uses the same data representation and training objective as our proposed solution, but it augment the training dataset by including randomly composed CAD command sequences (although the augmentation may be an invalid CAD sequence in few cases).

More details about these variants are described in Sec. D of the supplementary document.

Discussion of results. The quantitative results are report in Table 2, and more detailed CD scores are given in Table 4 of the supplementary document. In general, `Ours+Aug` (i.e., training with synthetic data augmentation) achieves the best performance, suggesting that randomly composed data can improve the network's generalization ability. The performance of `Alt-ArcMid` is similar to `Ours`. This means that

Figure 5. **Randomly generated 3D shapes** from our model (top) and l-GAN (bottom).

Method	COV↑	MMD↓	JSD↓
Ours	**78.13**	1.45	**3.76**
l-GAN	77.73	**1.27**	5.02

Table 3. **Shape generation measured under point-cloud metrics.** We use the metrics in l-GAN [6]. Both MMD and JSD are multiplied by 10^2. ↑: the higher the better, ↓: the lower the better.

middle-point representation is a viable alternative to represent arcs. Moreover, `Alt-Trans` performs slightly worse in terms of CD than `Ours` (e.g., see the green model in Fig. 4).

Perhaps more interestingly, while `Alt-Rel` has high parameter accuracy (ACC_{param}), even higher than `Ours`, it has a relatively large CD score and sometimes invalid topology: for example, the yellow model in the second row of Fig. 4 has two triangle loops intersecting with each other, resulting in invalid topology. This is caused by the errors of the predicted curve positions. In `Alt-Rel`, curve positions are specified with respect to its predecessor curve, and thus the error accumulates along the loop.

Lastly, `Alt-Regr`, not quantizing continuous parameters, suffers from larger errors that may break curcial geometric relations such as parallel and perpendicular edges (e.g., see the orange model in Fig. 4).

Cross-dataset generalization. We also verify the generalization of our autoencoder: we take our autoencoder trained on our created dataset and evaluate it on the smaller dataset provided in [48]. These datasets are constructed from different sources: ours is based on models from Onshape repository, while theirs is produced from designs in Autodesk Fusion 360. Nonetheless, our network generalizes well on their dataset, achieving comparable quantitative performance (see Sec. E in supplementary document).

4.2. Shape Generation

Next, we evaluate CAD model generation from latent vectors (described in Sec. 3.4). Some examples of our generated CAD models are shown in Fig. 1, and more results are
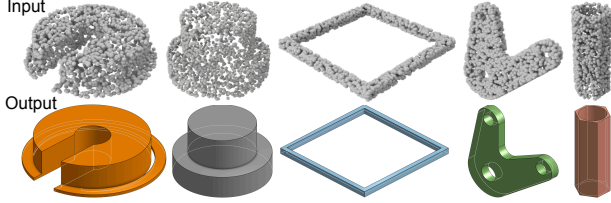
Figure 6. **CAD model reconstruction from point clouds**. (Top) input point clouds. (Bottom) reconstructed CAD models.
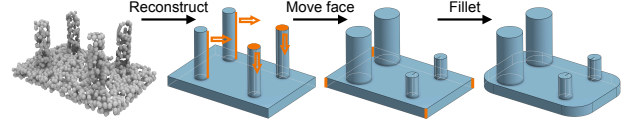

Figure 7. **User Editing.** Our reconstructed CAD model can be easily edited in any CAD tools. Here, the regions that undergo CAD operations are highlighted in orange color.

presented in Fig. 14 of the supplementary document.

Since there are no existing generative models for CAD designs, we choose to compare our model with *l-GAN* [6], a widely studied point-cloud 3D shape generative model. We note that our goal is *not* to show the superiority one over another, as the two generative models have different application areas. Rather, we demonstrate our model's ability to generate comparable shape quality even under the metrics for point-cloud generative models. Further, shapes from our model, as shown in Fig. 5, have much sharper geometric details, and they can be easily user edited (Fig. 7).

Metrics. For quantitative comparison with point-cloud generative models, we follow the metrics used in l-GAN [6]. Those metrics measure the discrepancy between two sets of 3D point-cloud shapes, the set $\mathcal{S}$ of ground-truth shapes and the set $\mathcal{G}$ of generated shapes. In particular, *Coverage (COV)* measures what percentage of shapes in $\mathcal{S}$ can be well approximated by shapes in $\mathcal{G}$. *Minimum Matching Distance (MMD)* measures the fidelity of $\mathcal{G}$ through the minimum matching distance between two point clouds from $\mathcal{S}$ and $\mathcal{G}$. *Jensen-Shannon Divergence (JSD)* is the standard statistical distance, measuring the similarity between the point-cloud distributions of $\mathcal{S}$ and $\mathcal{G}$. Details of computing these metrics are present in the supplement (Sec. G).

Discussion of results. Figure 5 illustrates some output examples from our CAD generative model and l-GAN. We then convert ground-truth and generated CAD models into point clouds, and evaluate the metrics. The results are reported in Table 3, indicating that our method has comparable performance as l-GAN in terms of the point-cloud metrics. Nevertheless, CAD models, thanks to their parametric representation, have much smoother surfaces and sharper geometric features than point clouds.

4.3. Future Applications

The CAD generative model can serve as a fundamental algorithmic block in many applications. While our work focuses on the generative model itself, not the downstream applications, here we discuss its use in two scenarios.

With the CAD generative model, one can take a point cloud (e.g., acquired through 3D scanning) and reconstruct a CAD model. As a preliminary demonstration, we use our autoencoder to encode a CAD model M into a latent vector c. We then leverage the PointNet++ encoder [35], training it

to encode the point-cloud representation of M into the same latent vector c. At inference time, provided a point cloud, we use PointNet++ encoder to map it into a latent vector, followed by our auotoencocder to decode into a CAD model. We show some visual examples in Fig. 6 and quan-titative results in the supplementary document (Table 6).

Furthermore, the generated CAD model can be directly imported into CAD tools for user editing (see Fig. 7). This is a unique feature enabled by the CAD generative model, as the user editing on point clouds or polygon meshes would be much more troublesome.

5. Discussion and Conclusion

Toward the CAD generative model, there are several limitations in our approach. At this point, we have considered three most widely used types of curve commands (line, arc, circle), but other curve commands can be easily added as well. For example, a cubic Bézier curve can be specified by three control points together with the starting point from the ending position of its predecessor. These parameters can be structured in the same way as described in Sec. 3.1. Other operations, such as revolving a sketch, can be encoded in a way similar to the extrusion command. However, certain CAD operations such as *fillet* operate on parts of the shape boundary, and thus they require a reference to the model's B-rep, not just other commands. To incorporate those commands in the generative model is left for future research.

Not every CAD command sequence can produce topologically valid shape. Our generative network cannot guarantee topological soundness of its output CAD sequences. In practice, the generated CAD command sequence rarely fails. The failure becomes more likely as the command sequence becomes quite long. We present and analyze some failure cases in Sec. F of the supplementary document, providing some fodder for future research.

In summary, we have presented DeepCAD, a deep generative model for CAD designs. Almost all previous 3D generative models produce discrete 3D shapes such as voxels, point clouds, and meshes. This work, to our knowledge, is the first generative model for CAD designs. To this end, we also introduce a large dataset of CAD models, each represented as a CAD command sequence.

References

[1] Autocad. https://www.autodesk.com/products/autocad. 2

[2] Fusion 360. https://www.autodesk.com/products/fusion-360. 2

[3] Onshape. http://http://onshape.com. 2, 5

[4] Onshape developer documentation. https://onshape-public.github.io/docs/. 5

[5] Onshape featurescript. https://cad.onshape.com/FsDoc/. 5

[6] Panos Achlioptas, Olga Diamanti, Ioannis Mitliagkas, and Leonidas Guibas. Learning representations and generative models for 3D point clouds. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 40–49, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. 1, 2, 6, 7, 8, 4

[7] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 214–223, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. 6, 2

[8] Ruojin Cai, Guandao Yang, Hadar Averbuch-Elor, Zekun Hao, Serge Belongie, Noah Snavely, and Bharath Hariharan. Learning gradient fields for shape generation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2020. 1, 2

[9] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *European Conference on Computer Vision*, pages 213–229. Springer, 2020. 2, 5

[10] Alexandre Carlier, Martin Danelljan, Alexandre Alahi, and Radu Timofte. Deepsvg: A hierarchical generative network for vector graphics animation. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 16351–16361. Curran Associates, Inc., 2020. 2, 5

[11] Zhiqin Chen, Andrea Tagliasacchi, and Hao Zhang. Bsp-net: Generating compact meshes via binary space partitioning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 45–54, 2020. 1, 2

[12] Zhiqin Chen and Hao Zhang. Learning implicit fields for generative shape modeling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5939–5948, 2019. 1, 2, 6, 7

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 2, 4

[14] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 2

[15] Yaroslav Ganin, Sergey Bartunov, Yujia Li, Ethan Keller, and Stefano Saliceti. Computer-aided design as language. *arXiv preprint arXiv:2105.02769*, 2021. 2

[16] Rohit Girdhar, David F Fouhey, Mikel Rodriguez, and Abhinav Gupta. Learning a predictable and generative vector representation for objects. In *European Conference on Computer Vision*, pages 484–499. Springer, 2016. 2

[17] Thibault Groueix, Matthew Fisher, Vladimir G Kim, Bryan C Russell, and Mathieu Aubry. A papier-mâché approach to learning 3d surface generation. pages 216–224, 2018. 1, 2, 6, 7

[18] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 5769–5779, USA, 2017. Curran Associates Inc. 6, 2

[19] Pradeep Kumar Jayaraman, Aditya Sanghi, Joseph Lambourne, Thomas Davies, Hooman Shayani, and Nigel Morris. Uv-net: Learning from curve-networks and solids. *arXiv preprint arXiv:2006.10211*, 2020. 2

[20] R. Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. Shapeassembly: Learning to generate programs for 3d shape structure synthesis. *ACM Transactions on Graphics (TOG), Siggraph Asia 2020*, 39(6):Article 234, 2020. 2

[21] Kacper Kania, Maciej Zięba, and Tomasz Kajdanowicz. Ucsgnet–unsupervised discovering of constructive solid geometry tree. *arXiv preprint arXiv:2006.09102*, 2020. 2

[22] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. 6

[23] Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. Abc: A big cad model dataset for geometric deep learning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019. 5, 1

[24] Joseph G Lambourne, Karl DD Willis, Pradeep Kumar Jayaraman, Aditya Sanghi, Peter Meltzer, and Hooman Shayani. Brepnet: A topological message passing system for solid models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12773–12782, 2021. 2

[25] Changjian Li, Hao Pan, Adrien Bousseau, and Niloy J. Mitra. Sketch2cad: Sequential cad modeling by sketching in context. *ACM Trans. Graph. (Proceedings of SIGGRAPH Asia 2020)*, 39(6):164:1–164:14, 2020. 2

[26] Jun Li, Kai Xu, Siddhartha Chaudhuri, Ersin Yumer, Hao Zhang, and Leonidas Guibas. Grass: Generative recursive autoencoders for shape structures. *ACM Transactions on Graphics (Proc. of SIGGRAPH 2017)*, 36(4):to appear, 2017. 2

[27] Yiyi Liao, Simon Donne, and Andreas Geiger. Deep marching cubes: Learning explicit surface representations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2916–2925, 2018. 2

[28] Christoph Lüscher, Eugen Beck, Kazuki Irie, Markus Kitza, Wilfried Michel, Albert Zeyer, Ralf Schlüter, and Hermann Ney. Rwth asr systems for librispeech: Hybrid vs attention. *Interspeech 2019*, Sep 2019. 4

[29] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4460–4470, 2019. 1, 2

[30] Kaichun Mo, Paul Guerrero, Li Yi, Hao Su, Peter Wonka, Niloy Mitra, and Leonidas J Guibas. Structurenet: Hierarchical graph networks for 3d shape generation. 2019. 1, 2

[31] Charlie Nash, Yaroslav Ganin, S. M. Ali Eslami, and Peter Battaglia. PolyGen: An autoregressive generative model of 3D meshes. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 7220–7229. PMLR, 13–18 Jul 2020. 1, 2, 4

[32] Wamiq Reyaz Para, Shariq Farooq Bhat, Paul Guerrero, Tom Kelly, Niloy Mitra, Leonidas Guibas, and Peter Wonka. Sketchgen: Generating constrained cad sketches. *arXiv preprint arXiv:2106.02711*, 2021. 2

[33] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 165–174, 2019. 1, 2

[34] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International Conference on Machine Learning*, pages 4055–4064. PMLR, 2018. 2

[35] Charles R Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *arXiv preprint arXiv:1706.02413*, 2017. 8

[36] Tim Salimans, Andrej Karpathy, Xi Chen, and Diederik P. Kingma. Pixelcnn++: A pixelcnn implementation with discretized logistic mixture likelihood and other modifications. In *ICLR*, 2017. 4

[37] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. Csgnet: Neural shape parser for constructive solid geometry. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5515–5523, 2018. 2

[38] Gopal Sharma, Difan Liu, Subhransu Maji, Evangelos Kalogerakis, Siddhartha Chaudhuri, and Radomír Měch. Parsenet: A parametric surface fitting network for 3d point clouds. In *European Conference on Computer Vision*, pages 261–276. Springer, 2020. 2

[39] Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T Freeman, Joshua B Tenenbaum, and Jiajun Wu. Learning to infer and execute 3d shape programs. *arXiv preprint arXiv:1901.02875*, 2019. 2

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. 2, 4, 5

[41] Homer Walke, R Kenny Jones, and Daniel Ritchie. Learning to infer shape programs using latent execution self training. *arXiv preprint arXiv:2011.13045*, 2020. 2

[42] Nanyang Wang, Yinda Zhang, Zhuwen Li, Yanwei Fu, Wei Liu, and Yu-Gang Jiang. Pixel2mesh: Generating 3d mesh models from single rgb images. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 52–67, 2018. 1, 2

[43] Xiaogang Wang, Yuelang Xu, Kai Xu, Andrea Tagliasacchi, Bin Zhou, Ali Mahdavi-Amiri, and Hao Zhang. Pie-net: Parametric inference of point cloud edges. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 20167–20178. Curran Associates, Inc., 2020. 2

[44] Xinpeng Wang, Chandan Yeshwanth, and Matthias Nießner. Sceneformer: Indoor scene generation with transformers. *arXiv preprint arXiv:2012.09793*, 2020. 2, 4

[45] K. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(1):21–40, 1985. 3

[46] Kevin Weiler. Topological structures for geometric modeling. 1986. 3

[47] Karl DD Willis, Pradeep Kumar Jayaraman, Joseph G Lambourne, Hang Chu, and Yewen Pu. Engineering sketch generation for computer-aided design. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2105–2114, 2021. 2

[48] Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. Fusion 360 gallery: A dataset and environment for programmatic cad reconstruction. *arXiv preprint arXiv:2010.02392*, 2020. 3, 5, 7, 1

[49] Jiajun Wu, Chengkai Zhang, Tianfan Xue, Bill Freeman, and Josh Tenenbaum. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling. In *Advances in Neural Information Processing Systems*, pages 82–90, 2016. 2

[50] Rundi Wu, Yixin Zhuang, Kai Xu, Hao Zhang, and Baoquan Chen. Pq-net: A generative part seq2seq network for 3d shapes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 829–838, 2020. 1, 2, 6

[51] Xianghao Xu, Wenzhe Peng, Chin-Yi Cheng, Karl DD Willis, and Daniel Ritchie. Inferring cad modeling sequences using zone graphs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6062–6070, 2021. 2

[52] Guandao Yang, Xun Huang, Zekun Hao, Ming-Yu Liu, Serge Belongie, and Bharath Hariharan. Pointflow: 3d point cloud generation with continuous normalizing flows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4541–4550, 2019. 1, 2

[53] Yaoqing Yang, Chen Feng, Yiru Shen, and Dong Tian. Foldingnet: Point cloud auto-encoder via deep grid deformation.

In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 206–215, 2018. 1, 2

Supplementary Document
DeepCAD: A Deep Generative Network for Computer-Aided Design Models

Rundi Wu   Chang Xiao   Changxi Zheng
Columbia University
{rundi, chang, cxz}@cs.columbia.edu

A. CAD dataset

We create our dataset by parsing the command sequences of CAD models in Onshape's online repository. Some example models from our dataset are shown in Fig. 9. Unlike other 3D shape datasets which have specific categories (such as chairs and cars), this dataset are mostly user-created mechanical parts, and they have diverse shapes.

Our dataset is derived from the ABC dataset [23], which contains several duplicate shapes. So for each shape in the test set, we find its nearest neighbor in the training set based on chamfer distance; we discard this test shape if the nearest distance is below a threshold.

We further examine the data distribution in terms of CAD command sequence length and the number of extrusions (see Fig. 8). Most CAD command sequences are no longer than 40 or use less than 8 extrusions, as these CAD models are all manually created by the user. A similar command length distribution is also reported in Fusion 360 Gallery [48].
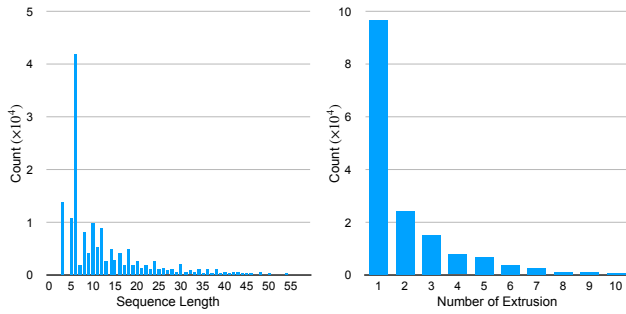

Figure 8. Statistics of CAD training data in terms of command sequence length (left) and number of extrusions (right).

B. Command Parameter Representation

Recall that our list of the full command parameters is $\boldsymbol{p}_i = [x, y, \alpha, f, r, \theta, \phi, \gamma, p_x, p_y, p_z, s, e_1, e_2, b, u]$ in Table 1. As described in Sec. 3.1.2, we normalize and quantize these parameters.

First, we scale every CAD model within a $2 \times 2 \times 2$ cube (without translation) such that all parameters stay bounded: the sketch plane origin (p_x, p_y, p_z) and the two-side extrusion distances (e_1, e_2) range in $[-1, 1]$; the scale of associated sketch profile s is within $[0, 2]$; and the sketch orientation (θ, ϕ, γ) have the range $[-\pi, \pi]$.

Next, we normalize every sketch profile into a unit square such that its starting point (i.e., the bottom-left point) locates at the center $(0.5, 0.5)$. As a result, the curve's ending position (x, y) and the radius r of a circle stay within $[0, 1]$. The arc's sweeping angle α by definition is in $[0, 2\pi]$.

Afterwards, we quantize all continuous parameters into 256 levels and express them using 8-bit integers.

For discrete parameters, we directly use their values. The arc's counter-clockwise flag f is a binary sign: 0 indicates clockwise arc and 1 indicates counter-clockwise arc. The CSG operation type $b \in \{0, 1, 2, 3\}$ indicates *new body*, *join*, *cut* and *intersect*, respectively. Lastly, the extrusion type $u \in \{0, 1, 2\}$ indicates *one-sided*, *symmetric* and *two-sided*, respectively.

C. Network Architecture and Training Details

Autoencoder. Our Transformer-based encoder and decoder are both composed of four layers of Transformer blocks, each with eight attention heads and a feed-forward dimension of 512. We adopt standard layer normalization and a dropout rate of 0.1 for each Transformer block.

The last Transformer block in the decoder is followed by two separate linear layers, one for predicting command type (with weights $W_1 \in \mathbb{R}^{256 \times 6}$), and another for predicting command parameters (with weights $W_2 \in \mathbb{R}^{256 \times 4096}$). The output, a 4096-dimensional vector, from the second linear layer is further reshaped into a matrix of shape 16×256, which indicates each of the total 16 parameters.

Latent-GAN. Section 3.4 describes the use of latent-GAN technique on our learned latent space for CAD generation. In our GAN model, the generator and discriminator are both MLP networks, each with four hidden layers. Every hidden layer has a dimension of 512. The input dimension (or the noise dimension) is 64, and the output dimension is 256.
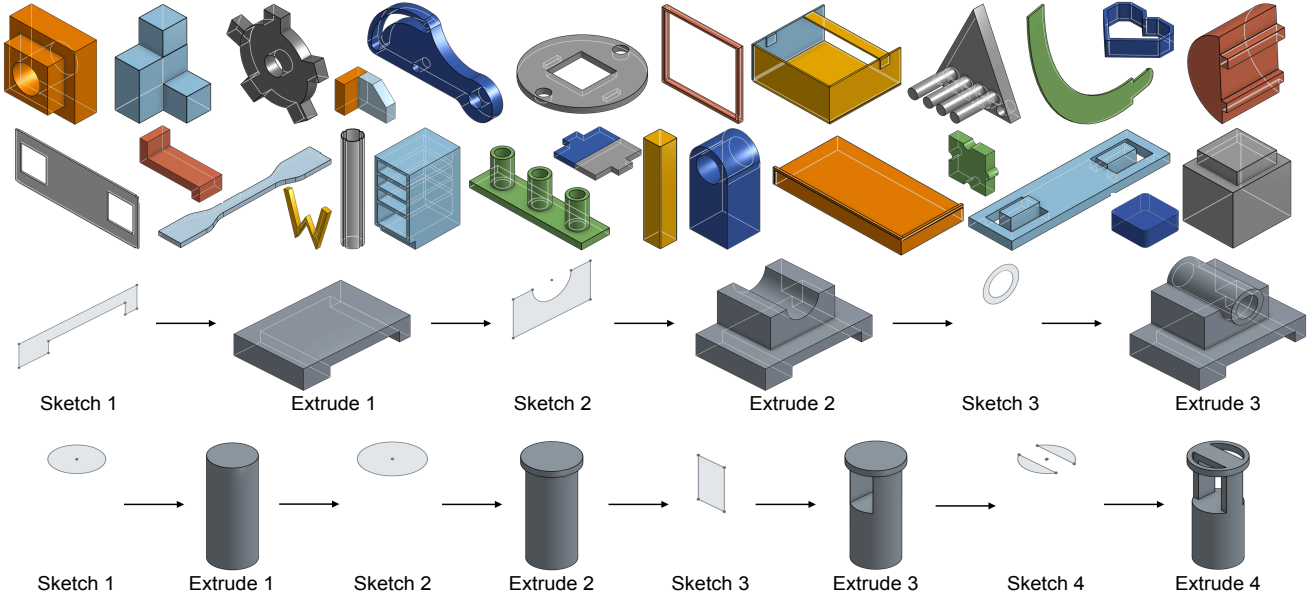
Figure 9. **Our CAD dataset**. Top two rows: examples of CAD models in our dataset. Bottom two rows: examples of CAD construction sequences.

We use WGAN-gp strategy [7, 18] to train the network: the number of critic iterations is set to 5 and the weight factor for gradient penalty is set to 10. The training lasts for $200,000$ iterations with a batch size of 256. In this process, Adam optimizer is used with a learning rate of 2×10^{-4} and $\beta_1 = 0.5$.

D. Autoencoding CAD models

Comparison methods. Here we describe in details the variants of our method used in Sec. 4.1 for comparison.

`Alt-Rel` represents curve positions relative to the position of its previous curve in the loop. As a result, the ending positions of a line and a arc and the center of a circle differ from those in our method, but the representation of other curve parameters (i.e., α, f, r in Table 1) stays the same.

`Alt-Trans` includes in the extrusion command the starting position (s_x, s_y) of the loop, in addition to the origin of the sketch plane. The origin (p_x, p_y, p_z) is in the world frame of reference; the loop's starting position (s_x, s_y) is described in the local frame of the sketch plane. In our proposed approach, however, we translate the sketch plane's origin to the loop's starting position. Thereby, there is no need to specify the parameters (s_x, s_y) explicitly.

`Alt-ArcMid` specifies an arc using its ending and middle positions. As a result, the representation of an arc becomes into (x, y, m_x, m_y), where (x, y) indicates the ending position (as in our method), but (m_x, m_y) is used to indicate the arc's middle point.

`Alt-Regr` regresses all parameters of the CAD commands using the standard mean-squared error in the loss function. The Cross-Entropy loss for discrete parameters

(such as command types) stays the same as our proposed approach. But in this variant, continuous parameters are not quantized, although they are still normalized into the range $[-1, 1]$ in order to balance the mean-squared errors introduced by different parameters.

`Ours+Aug` includes randomly composed CAD command sequences in its training process. This is a way of data augmentation. When we randomly choose a CAD model from the dataset during training, there is 50% chance that the sampled CAD sequence will be mixed with another randomly sampled CAD sequence. The mixture of the two CAD command sequences is done by randomly switching one or more pairs of sketch and extrusion (in their commands). CAD sequences that contain only one pair of sketch and extrusion are not involved in this process.

Full statistics for CD scores. In Table 4, we report the mean, trimmed mean, and median chamfer distance (CD) scores for our CAD autoencoding study. "Trimmed mean"

Method	mean CD	trimmed mean CD	median CD
`Ours+Aug`	**6.14**	**0.974**	**0.752**
`Ours`	7.16	1.08	0.787
`Alt-ArcMid`	6.90	1.09	0.790
`Alt-Trans`	7.14	1.09	0.792
`Alt-Rel`	9.24	1.38	0.863
`Alt-Regr`	12.61	3.87	2.14

Table 4. Mean, trimmed mean and median chamfer distances for shape autoencoding. Numerical values are multiplied by 10^3.
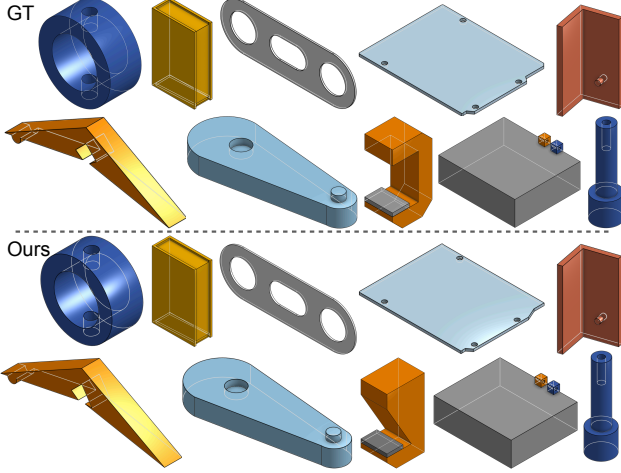
GT

Ours

Figure 10. Shape autoencoding results on Fusion 360 Gallery [48] test data. The autoencoder is trained using our own dataset.

ACC$_{cmd}$ ↑	ACC$_{param}$ ↑	median CD↓	Invalid Ratio↓
97.90	96.45	0.796	1.62

Table 5. Quantitative evaluation for shape autoencoding on Fusion 360 Gallery [48] test data. The model is only trained on our proposed dataset. ↑: the higher the better, ↓: the lower the better.

CD is computed by removing 10% largest and 10% smallest scores. The mean CD scores are significantly higher than the trimmed mean and median CD scores. This is because the prediction of CAD sequence in some cases may be sensitive to small perturbations: a small change in command sequence may lead to a large change of shape topology and may even invalidate the topology (e.g., the gray shape in Fig. 11). Those cases happen rarely, but when they happen, the CD scores become significantly large. It is those outliers that make the mean CD scores much higher.

Accuracies of individual parameter types. We also examine the accuracies for individual types of parameters. The accuracy is defined in Sec. 4.1, and the results are shown in Fig. 13. While all the parameter are treated equally in the loss function, their accuracies have some differences. Most notably, the recovery of arc's sweeping angle α has lower accuracy than other parameters. By examining the dataset, we find that the values of sweeping angle α span over its value range (i.e., $[0, 2\pi]$) more evenly than other parameters, but the arc command is much less frequently used than other commands. Thus, in comparison to other parameters, it is harder to learn the recovery of the arc sweeping angle.

E. Generalization on Fusion 360 Gallery [48]

To validate the generalization ability of our autoencoder, we perform a cross-dataset test. In particular, for shape autoencoding tasks, we take the model trained on our proposed
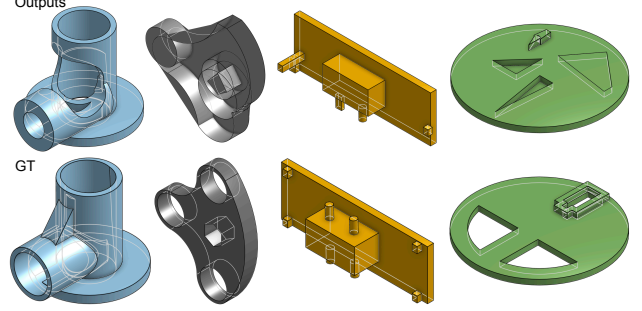
Outputs

GT

Figure 11. Failure examples in shape autoencoding. Top: our reconstructed CAD outputs. Bottom: ground-truth CAD models.
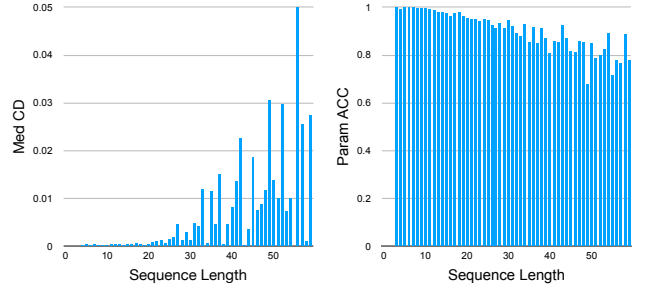

Figure 12. Quantitative metrics for shape autoencoding w.r.t. CAD sequence length. Left: median chamfer distance (the lower the better). Right: parameter accuracy (the higher the better).
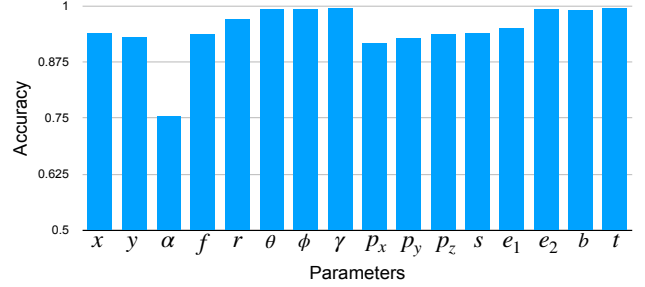

Figure 13. Accuracies for individual parameter types.

Method	ACC$_{cmd}$ ↑	ACC$_{param}$ ↑	median CD↓	Invalid Ratio↓
Ours	85.95	74.22	10.30	12.08
Ours-noise	84.65	74.23	10.44	13.82

Table 6. Quantitative results for CAD reconstruction from point clouds. Ours-noise corresponds to noisy inputs (uniform noise in $[-0.02, 0.02]$ along normal direction). We use the same metrics as in autoencoding task; ACC$_{cmd}$ and ACC$_{param}$ are both multiplied by 100%, and CD is multiplied by 10^3.

dataset and evaluate it using a different dataset provided by Fusion 360 Gallery [48]. These two datasets are constructed from different sources: ours is based on models from Onshape repository, whereas theirs is created from designs in Autodesk Fusion 360. The qualitative and quantitative results are shown in Fig. 10 and Table 5, respectively, showing that our trained model performs well on shape distributions
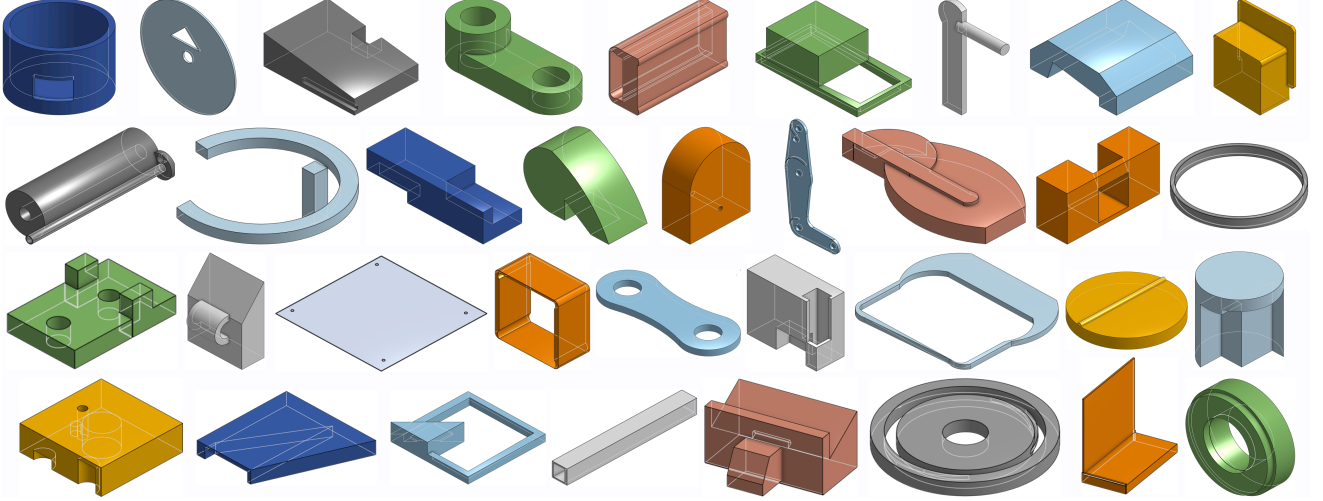
Figure 14. A gallery of our generated CAD models.

that are different from the training dataset.

F. Failure Cases

Not every CAD command sequence is valid. Our method is more likely to produce invalid CAD commands when the command length becomes long. Figure 11 shows a few failed results. The produced gray shape has invalid topology, and the yellow shape suffers from misplacement of small sketches. Figure 12 plots the median CD scores and the parameter accuracies with respect to CAD command sequence length. The difficulties for generating long-sequence CAD models are twofold. As the CAD sequence becomes longer, it is harder to ensure valid topology. Meanwhile, as shown in Fig. 8, the data distribution in terms of the sequence length has a long tail; the dataset provides much more short sequences than long sequences. This data imbalance may cause the network model to bias toward short sequences.

G. Metrics for Shape Generation

We follow the three metrics used in [6] to evaluate the quality of our shape generation. In [6], these metrics are motivated for evaluating the point-cloud generation. Therefore, for computing these metrics for CAD models, we first convert them into point clouds. Then, these metrics are defined by comparing a set of reference shapes $\mathcal{S}$ with a set of generated shapes $\mathcal{G}$.

Coverage (COV) measures the diversity of generated shapes by computing the fraction of shapes in the reference set $\mathcal{S}$ that are matched by at least one shape in the generated set $\mathcal{G}$. Formally, COV is defined as

$$\text{COV}(\mathcal{S}, \mathcal{G}) = \frac{|\{\arg\min_{Y \in \mathcal{S}} d^{\text{CD}}(X, Y) | X \in \mathcal{G}\}|}{|\mathcal{S}|}, \quad (6)$$

where $d^{\text{CD}}(X, Y)$ denote the chamfer distance between two point clouds X and Y.

Minimum matching distance (MMD) measures the fidelity of generated shapes. For each shape in the reference set $\mathcal{S}$, the chamfer distance to its nearest neighbor in the generated set $\mathcal{G}$ is computed. MMD is defined as the average over all the nearest distances:

$$\text{MMD}(\mathcal{S}, \mathcal{G}) = \frac{1}{|\mathcal{S}|} \sum_{Y \in \mathcal{S}} \min_{X \in \mathcal{G}} d^{\text{CD}}(X, Y). \quad (7)$$

Jensen-Shannon Divergence (JSD) is a statistical distance metric between two data distributions. Here, it measures the similarity between the reference set $\mathcal{S}$ and the generated set $\mathcal{G}$ by computing the marginal point distributions:

$$\text{JSD}(P_{\mathcal{S}}, P_{\mathcal{G}}) = \frac{1}{2} D_{\text{KL}}(P_{\mathcal{S}} || M) + \frac{1}{2} D_{\text{KL}}(P_{\mathcal{G}} || M), \quad (8)$$

where $M = \frac{1}{2}(P_{\mathcal{S}} + P_{\mathcal{G}})$ and D_{KL} is the standard KL-divergence. $P_{\mathcal{S}}$ and $P_{\mathcal{G}}$ are marginal distributions of points in the reference and generated sets, approximated by discretizing the space into 28^3 voxel grids and assigning each point from the point cloud to one of them.

Since our full test set is relatively large, we randomly sample a reference set of 1000 shapes and generate 3000 shapes using our method to compute the metric scores. To reduce the sampling bias, we repeat this evaluation process for three times and report the average scores.