`CAD-Recode`: Reverse Engineering CAD Code from Point Clouds

Danila Rukhovich[1]
danila.rukhovich@uni.lu
Elona Dupont[1]
elona.dupont@uni.lu
Dimitrios Mallis[1]
dimitrios.mallis@uni.lu
Kseniya Cherenkova[1,2]
kseniya.cherenkova@uni.lu
Anis Kacem[1]
anis.kacem@uni.lu
Djamila Aouada[1]
djamila.aouada@uni.lu

[1]SnT, University of Luxembourg [2]Artec3D, Luxembourg

Abstract

Computer-Aided Design (CAD) models are typically constructed by sequentially drawing parametric sketches and applying CAD operations to obtain a 3D model. The problem of 3D CAD reverse engineering consists of reconstructing the sketch and CAD operation sequences from 3D representations such as point clouds. In this paper, we address this challenge through novel contributions across three levels: CAD sequence representation, network design, and training dataset. In particular, we represent CAD sketch-extrude sequences as Python code. The proposed **`CAD-Recode`** *translates a point cloud into Python code that, when executed, reconstructs the CAD model. Taking advantage of the exposure of pre-trained Large Language Models (LLMs) to Python code, we leverage a relatively small LLM as a decoder for* `CAD-Recode` *and combine it with a lightweight point cloud projector.* `CAD-Recode` *is trained on a procedurally generated dataset of one million CAD sequences.* `CAD-Recode` *significantly outperforms existing methods across the DeepCAD, Fusion360 and real-world CC3D datasets. Furthermore, we show that our CAD Python code output is interpretable by off-the-shelf LLMs, enabling CAD editing and CAD-specific question answering from point clouds.*

1. Introduction

Computer-Aided Design (CAD) modeling is the standard approach for designing manufactured objects, ranging from furniture to mechanical components [3, 30]. However, creating a 3D CAD model is a time-consuming task that demands specialized expertise, as the model must not only capture the object's shape but also maintain its functional requirements—commonly referred to as the *design intent* [4, 27]. To streamline this process, *3D CAD reverse engineering* aims at generating CAD models directly from 3D scanned objects, offering a faster and more accessible pathway to CAD creation [17].

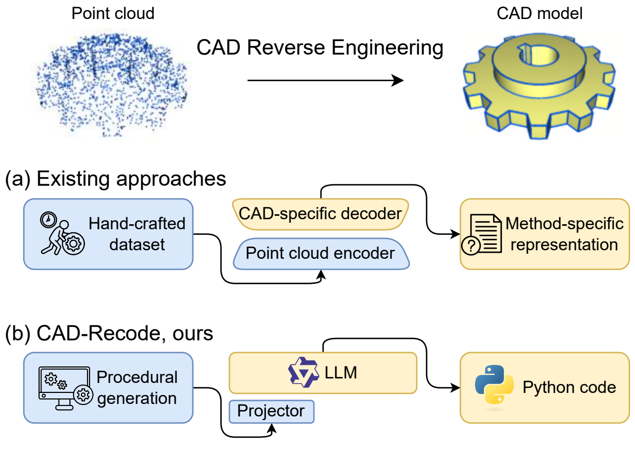


Figure 1. 3D CAD reverse engineering is a process of converting a point cloud into a CAD model (top). Existing methods are constrained by the use of method-specific CAD representations and limited hand-crafted training datasets (a). On the contrary, `CAD-Recode` employs a pre-trained LLM with a lightweight projector that translates point clouds into executable Python code and is trained on a procedurally generated dataset (b).

Automated 3D CAD reverse engineering has a long history in the fields of computer vision and graphics [13, 43], with goals evolving alongside advancements in the field. These objectives have progressed from identifying CAD parts in 3D point clouds [43] to predicting the sequence of steps a designer may take to recreate a 3D scanned object in CAD software [21, 34]. This latter goal is particularly appealing, as it aims not only to produce a final CAD parametric model but also to capture the design steps behind it, enabling further editing within CAD software [21, 45]. In CAD software, designers typically construct their CAD model as feature-based design sequences, where a sequence of 2D sketches is transformed into 3D objects via operations such as extrusion and revolution [47, 50]. Following the release of large CAD datasets [7, 22, 47], recent works have focused on learning feature-based CAD sequences from input point clouds, specifically as *sketch-extrude* sequences [12, 21, 28, 33, 39, 45, 48]. As depicted in Figure 1(a), although varying in methodology, these approaches share a common pipeline: (1) crafting a CAD sketch-extrude sequence representation, (2) converting raw CAD data [22, 47] into this format, and (3) training dedicated neural networks to output these representations from input point clouds.

Despite recent advances in feature-based CAD reverse engineering, key limitations constrain the broader applicability of existing approaches. Firstly, existing methods often use customized CAD representations, such as custom CAD languages [12, 21, 34, 48] or simplified extrusion shapes [28, 39, 45], to facilitate model training. These representations are difficult to interpret, require post-processing to be compatible with CAD tools, and restrict design capabilities to basic operations. Secondly, these approaches typically rely on designing networks that output language-like CAD representations [12, 21] and training them from scratch. This requires the networks to learn not only the geometry of the point clouds, but also the syntax of the CAD sequence representation.

In this paper, we pose the following question: *In view of the recent breakthrough performance of Large Language Models (LLMs), how can their advanced language understanding be leveraged for CAD reverse engineering?*

To address this question, we base our approach on three key observations: (1) LLMs can generate valid Python code [36, 55], (2) modern CAD software increasingly supports modeling through Python scripting [10], and (3) recent efforts have shown that LLMs can be fine-tuned to process point clouds [49, 61]. As shown in Figure 1(b), *we propose `CAD-Recode`, a solution for CAD reverse engineering by fine-tuning an LLM to map input point clouds into CAD sketch-extrude sequences represented as Python code.* In particular, instead of crafting a CAD representation, we base our representation on the existing Python CadQuery library [10]. This code-based representation is not only interpretable but also inherently allows for incorporating modular CAD features and design practices such as reusing design elements and abstracting low-level design steps (*e.g.* 3D box to represent a four-line sketch of a square and its extrusion). To learn the mapping between point clouds and CAD Python code, we fine-tune a pre-trained LLM, Qwen2-1.5B [55], augmented with a lightweight, trainable point cloud projector. To train `CAD-Recode`, a potential approach could be using existing sketch-extrude datasets [47, 48] and converting them to Python CadQuery code. However, these datasets are limited in size and design features included due to the efforts required to convert their original proprietary representation into one that is suitable for learning. As a solution, we propose a *procedurally* generated training dataset composed of one million CAD sketch-extrude sequences as Python CadQuery code. This dataset consists of CadQuery Python scripts generated following predefined heuristics with randomized parameter selection. The execution of each generated script results in a parametric CAD model. Unlike existing CAD datasets, this procedurally generated dataset provides an alternative for learning the mapping between point clouds and CAD sketch-extrude sequences in Python code, with full control over the design features, patterns and dataset size included during training. Our contributions can be summarized to:

- A CAD sketch-extrude sequence representation in Python code using CadQuery [10] for CAD reverse engineering.
- `CAD-Recode`, the first LLM-based CAD reverse engineering model designed to predict CAD Python code from point clouds. The model, consisting of a pre-trained LLM and a point cloud projector is trained end-to-end to generate code that reconstructs the input geometry.
- A one million procedurally generated training dataset of CAD sketch-extrude sequences as CadQuery Python code. This provides precise control over dataset size, features, and design patterns included during training, resulting in significant performance improvement. We will make this dataset publicly accessible.
- Extensive experiments on three publicly available datasets show that `CAD-Recode` achieves substantial improvements over state-of-the-art methods in CAD reverse engineering. Moreover, we show that `CAD-Recode`, when operating on point clouds and generating CAD code, can be integrated with an off-the-shelf LLM to perform CAD Question Answering (CAD-QA) and CAD editing from point clouds.

2. Related Works

LLM, Point Cloud and CAD: Recent works have explored integrating point clouds with LLMs for various tasks, including 3D generation [56, 62], captioning [16, 18, 49], and question answering [6, 19]. These approaches typically employ complex point cloud encoders, either aligning with CLIP embeddings [31, 53, 54, 61, 64] or directly with LLM feature spaces [49]. Such methods require two-stage training: first pre-training the point cloud encoder, then fine-tuning with the LLM through instruction-based prompts.

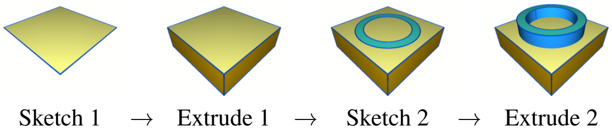
In parallel, recent works have started exploring LLMs' capabilities in a range of CAD-related tasks. Reparam-CAD [23] infers shape variations from parametric models and text descriptions, while CADTalk [60] generates semantic descriptions of CAD parts. The works in [1, 2] focus on the generation of CAD models from text using LLMs, and SGP-Benchmark [38] evaluates LLMs' understanding of CAD sketch-extrude sequences using CAD-specific question answering. While Img2CAD [57] attempts CAD reverse engineering from images using GPT-4V [36], it still requires a separate transformer for parameter inference. In contrast, `CAD-Recode` introduces the first approach for point cloud to CAD reconstruction combining point clouds with the sequence modeling capabilities of pre-trained LLMs.

CAD Reverse Engineering from Point Cloud: CAD reverse engineering aims to reconstruct parametric CAD models from 3D shapes (*e.g.*, meshes or point clouds) in a compatible representation with CAD software. A key challenge lies in the choice of this representation. A line of works attempts to address sub-problems of the CAD reverse engineering task by focusing on parameter estimation for edges and surface primitives [8, 9, 15, 26, 32, 42, 46, 63] or reconstructing B-Rep construction history [11, 24, 47, 50]. In order to obtain a representation that is closer to CAD modelling, several methods [14, 20, 29, 58, 59] use Constructive Solid Geometry (CSG), representing models as compositions of 3D primitives and Boolean operations. While this enables reconstruction, the CSG representation diverges from modern CAD workflows [50].

Recent works have adopted the more CAD-aligned sketch-extrude representation, introduced by DeepCAD [48] for CAD generation [51, 52] or predicting extrusion cylinder [39, 45]. Considering the sequential nature of sketch-extrude operations, methods have explored a template-based approach [25] given a rounded voxel input representation. Furthermore, transformer architectures have been investigated for both autoregressive [21] and non-autoregressive [12, 48] prediction of sketch-extrude sequences from point clouds. The work in [34] combines a lightweight pre-trained language model [40] with a point cloud encoder using a diffusion-based approach. Alternative methods using self-supervised [28] or unsupervised [29] learning still face integration challenges due to their non-standard sketch representations (*e.g.*, signed distance functions). In contrast to these approaches that require full parameter learning of specialized networks for both CAD geometry and representation syntax, we leverage pre-trained LLMs that have been exposed to programming patterns through large-scale training on code repositories. Our method outputs Python code using the CadQuery library [10] that is directly executable and can easily be interpreted. Additionally, we address the data limitation through automated generation of a large-scale training dataset, enabling full control over design features included during training.

3. CAD Representation as Code

Modern feature-based CAD modeling relies on sequences of 2D sketches and operations to create 3D models. Designers first draw geometric primitives (lines, arcs, circles) on a selected plane, then apply operations like extrusion or revolution to generate 3D geometry [50]. As depicted in Figure 2(a), we focus on sketch-extrusion sequences, a fundamental CAD modeling pattern widely adopted in previous works [21, 48, 51]. Below, we present our CAD representation, highlighting its advantages over existing language-like encodings, and describe our procedurally generated training data.



(a) Sketch-Extrude sequence

```
SOL Line -50 -50 Line 50 50 Line 50 -50 Line -25
-50 Ext. 0 0 1 0 0 0 14 0 NewBody SOL Circle 0 0 40
SOL Circle 0 0 30 Ext. 0 0 1 0 0 0 -10 0 Union
```

(b) DeepCAD representation

```
import cadquery as cq
w = cq.Workplane('XY')
w.box(100,100,14).union(
  w.sketch().circle(40).circle(30,mode='s')
  .finalize().extrude(-10))
```

(c) Our CadQuery representation

Figure 2. Sketch-extrude sequence (top) in DeepCAD representation (middle) and our CadQuery code (bottom).

3.1. CadQuery Code

Recent approaches in CAD language modeling [12, 21, 33, 48, 51] encode sketch-extrude sequences as numerical vectors representing features and their parameters as shown in Figure 2(b). However, this representation constrains the modeling to specific CAD practices, lacks interpretability, and requires post-processing for CAD kernel compatibility. We propose using CadQuery [10] Python code to represent sketch-extrude sequences for CAD reverse engineering, offering the following advantages:

Modularity of CAD Features and Design Practices: Existing language-based CAD reverse engineering methods rely on custom representations of low-level geometric features (lines, arcs, circles) for sketch construction [41, 48]. This approach inherently limits the range of implementable features and design practices. In contrast, CadQuery provides comprehensive built-in CAD functionality, encompassing both low-level features and higher-level geometries like cylinders and boxes as shown in Figure 2(c). Furthermore, its programmatic nature enables variable reuse and code modularity. This allows reusing common design features or practices across models, as illustrated by the shared center coordinates across two circles in Figure 2 (c). This representation naturally accommodates diverse CAD features and design practices without requiring complex custom encodings or post-processing steps.

Interpretability and LLM Compatibility: The proposed representation, based on Python and CadQuery syntax, presents an alternative to abstract numerical encodings with improved interpretability. Its code-based format facilitates model editing both programmatically and through CAD software. Importantly, this representation aligns with pre-trained LLMs' demonstrated proficiency in Python code generation and manipulation. Indeed, state-of-the-art proprietary LLMs like GPT-4 [36] achieve over 90% accuracy on the Python code HumanEval benchmark [5], while even lightweight open-source models such as Qwen2-1.5B [55] show promising code generation capabilities. Hence, this code representation facilitates fine-tuning of pre-trained LLMs for the specific task of reverse engineering point clouds into CAD Python code and opens the doors for new capabilities with off-the-shelf LLMs.

3.2. Procedurally Generated Training Dataset

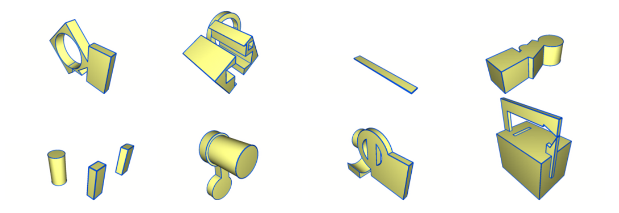


Figure 3. Examples of procedurally generated CAD models.

The training of current CAD sketch-extrude reverse engineering methods [12, 21, 33, 34, 48] predominantly rely on datasets collected from CAD model repositories [7, 22, 47]. Considerable efforts are required to parse the CAD models from their original proprietary representations to a suitable one for deep learning [47, 48]. As a result, existing datasets are restricted not only in scale, but also in control over the design features and patterns included in training.

To address these limitations, we propose to procedurally generate a training dataset of one million CAD models in the form of sketch-extrude sequences written in Python CadQuery [10] code. Our proposed pipeline randomly generates sketch and CAD operation parameters guided by topological and geometrical heuristics to ensure control over the amount of generated models and the features in the generated codes. The algorithm outlining the steps of this generation pipeline is provided in the supplementary materials along with further statistical analysis of the generated dataset. The modularity of CAD features is incorporated by utilizing both low-level primitives (*i.e.* circles, lines, and arcs) and their abstractions (*i.e.* boxes, cylinders, and rectangles) as well as reusing design elements within the generated sequences. In this work, we focus on some aspects of modularity (*i.e.*, reusing point coordinates, extrusion planes, and abstracting basic shapes such as boxes and cylinders). Further modularity features (*e.g.*, reusing functions corresponding to arbitrary CAD parts, additional CAD operations) can also be integrated in the future. Note that although our generated dataset does not include sequences from human-designed CAD models, it offers significant control over the features and design patterns to which the network is exposed during training. Examples of generated CAD models are shown in Figure 3.

4. `CAD-Recode`

Building on the introduced CAD code representation and generated training dataset outlined in the previous section, this section introduces `CAD-Recode`, our proposed model for predicting CAD sketch-extrude sequences as code from input 3D point clouds. We formalize the problem of CAD code prediction, describe the architecture of `CAD-Recode`, and detail its training and inference processes.

4.1. Problem Statement

Let us denote the set of all possible code strings as Σ^*, where each code string is composed of elements from the finite set of alphanumeric characters and operators in the programming language Σ. Let $\phi_{\text{syn}} : \Sigma^* \to \{\texttt{True}, \texttt{False}\}$ represent the syntactic validation function for Python programming rules (*e.g.*, variable declarations, expression syntax, and statement structure), and $\phi_{\text{cad}} : \Sigma^* \to \{\texttt{True}, \texttt{False}\}$ denote the validation function for CAD-specific rules. The latter includes the syntactic validity of the code w.r.t. to the CAD library, *i.e.* CadQuery [10], and the geometric validity of the reconstructed model from the code (*e.g.*, an extrusion can only be applied on a closed loop of 2D primitives, a circle radius cannot be negative). An executable valid CAD code can be formally described by a code string $C \in \mathcal{C}$, where

$$\mathcal{C} = \{w \in \Sigma^* \mid \phi_{\text{syn}}(w) \wedge \phi_{\text{cad}}(w)\} ,$$

represents the set of all valid CAD codes. This formulation ensures that any code string w in $\mathcal{C}$ satisfies both the syntactic requirements of Python (ϕ_{syn}) and the CAD code validation rules (ϕ_{cad}). Let $\mathbf{P} = \{\mathbf{p}_i\}_{i=1}^{n} \in \mathbb{R}^{n\times 3}$ denote an input point cloud, where each point $\mathbf{p}_i \in \mathbb{R}^3$ represents 3D Cartesian coordinates. The objective of `CAD-Recode` is to learn a mapping

$$\Psi : \mathbb{R}^{n\times 3} \to \mathcal{C}, \quad C = \Psi(\mathbf{P}) ,$$

that maps the input point cloud to a valid CAD code $C \in \mathcal{C}$ such that the code, when executed, produces a CAD model geometrically approximating the input point cloud $\mathbf{P}$. Note that the CAD code execution results in a

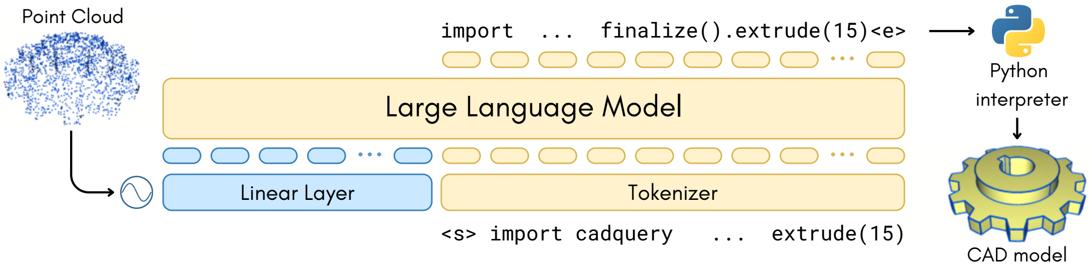


Figure 4. Overview of `CAD-Recode`. The pipeline comprises two parts: (1) a point cloud projector (marked blue) (2) a fine-tuned pre-trained LLM (yellow). An input point cloud is processed using (1), and outputs are then passed to an LLM (2), which predicts a CAD sketch-extrude sequence in the form of executable Python code.

Boundary-Representation (B-Rep) [24]. Unlike meshes or point clouds, B-Rep is a parametric representation of the CAD model's geometry, enabling seamless integration into modern CAD software and allowing for further modifications. The goal of `CAD-Recode` is to infer the CAD code describing the design steps of the CAD model, that when executed results in a B-Rep.

4.2. Proposed Model Architecture

`CAD-Recode` builds on pre-trained LLMs and their prior exposure to Python code, augmenting these with point cloud processing capabilities and CAD-specific Python code knowledge. As shown in Figure 4, its architecture consists of two components: (1) a point cloud projector mapping the 3D point cloud into learnable tokens, and (2) a pre-trained LLM-based auto-regressive CAD code decoder.

Point Cloud Projection Module: `CAD-Recode` introduces a lightweight projection module Ψ_p that directly maps a dense point cloud $\mathbf{P} \in \mathbb{R}^{n \times d_p}$, where $d_p = 3$ corresponds to the dimension of point coordinates, into a sequence of $n_p \ll n$ query tokens $\mathbf{Q}_p = [\mathbf{q}_p^1, \ldots, \mathbf{q}_p^{n_p}] \in \mathbb{R}^{n_p \times d_q}$, of embedding dimension d_q. The point cloud projector, trained in an end-to-end manner with the CAD code decoder module, consists of three simple components: (1) furthest point sampling to downsample the input point clouds to n_p points, (2) Fourier positional encoding [62] of coordinates, and (3) a linear layer projecting the encoded coordinatesinto $\mathbf{Q}_p$.

LLM as CAD Code Decoder: Our CAD code decoder, denoted as Ψ_{LLM}, adapts a pre-trained LLM for the specific task of CAD code generation. We leverage the Qwen2-1.5B model [55] as our LLM backbone, chosen for its balanced trade-off between model capacity and computational requirements. The decoder's input consists of point query tokens $\mathbf{Q}_p$ from the point cloud projector, augmented with n_t code tokens $\mathbf{Q}_t \in \mathbb{R}^{n_t \times d_q}$ obtained by tokenizing the input code as in [55]. The complete input sequence is denoted as $[\mathbf{Q}_p; \mathbf{Q}_t] \in \mathbb{R}^{(n_p+n_t) \times d_q}$, where semicolon indicates concatenation along the sequence dimension. The LLM decoder generates the CAD code sequence through next-token prediction. As in [55], each predicted token is mapped to a symbol from the vocabulary Σ, which includes alphanumeric characters and operators.

Overall, `CAD-Recode` repurposes the LLM's sequence modeling capabilities for the specialized task of translating geometric point clouds into executable CAD code.

4.3. Training and Inference Details

Training Strategy: Our training process consists of a single stage. The model operates on query tokens of dimension $d_q = 1536$ and processes input point clouds downsampled to $n_p = 256$ points. Gaussian noise with mean zero and standard deviation of 0.01 is added to the coordinates of the input points with a probability of 0.5 per model. The network is trained on the procedurally generated CAD codes, hence exposed to the CAD features and design practices that were included in the algorithm. The training objective minimizes the Negative Log-Likelihood (NLL) of the target CAD code sequence, using special tokens ($<s>$ and $<e>$) to demarcate sequence boundaries. The point cloud projector Ψ_p learns geometric features from scratch, while the pre-trained decoder Ψ_{LLM} is fine-tuned for CAD code generation.

Inference Strategy: At inference time, the point cloud projector Ψ_p processes the input point cloud $\mathbf{P}$ to generate query tokens $\mathbf{Q}_p$, which are then fed to the decoder along with the start token $<s>$. The model autoregressively generates CAD code tokens until producing a complete code sequence C ending with token $<e>$. Following [21], we employ a test-time sampling approach where we generate ten distinct CAD code candidates, each from a different sampling of the input point cloud. For each candidate, we sample points from the predicted CAD model and compute the Chamfer distance w.r.t. the input point cloud. The candidate with the minimum Chamfer distance is selected as the final output. This verification step effectively favors executable CAD code solutions that are geometrically consistent w.r.t. the input point cloud.

Method	Train Dataset		DeepCAD Test Set				Fusion360 Test Set			
	Name	Size	Mean CD↓	Med. CD↓	IoU↑	IR ↓	Mean CD↓	Med. CD↓	IoU↑	IR↓
DeepCAD [48]	DeepCAD	160k	42.5	9.64	46.7	7.1	330	89.2	39.9	25.2
PrismCAD [25]	DeepCAD	127k	–	4.28	72.1	16.2	–	4.75	65.3	18.0
Point2Cyl [45]	DeepCAD	35k	–	4.27	73.8	3.9	–	4.18	67.5	3.2
HNC-CAD [52]	DeepCAD	125k	–	8.64	65.3	5.6	–	36.8	63.5	7.3
MultiCAD [33]	DeepCAD	160k	–	8.09	–	11.5	–	42.2	–	16.5
TransCAD [12]	DeepCAD	140k	32.3	4.51	65.5	1.1	78.6	33.4	60.2	2.4
CAD-Diffuser [34]	DeepCAD	160k	–	3.02	74.3	1.5	–	3.85	63.2	1.7
CAD-SIGNet [21]	DeepCAD	160k	3.43	0.28	77.6	0.9	7.37	0.48	65.6	1.6
CAD-Recode	DeepCAD	160k	0.89	0.20	86.2	**0.0**	1.77	0.30	75.6	**0.0**
CAD-Recode	Ours	1M	**0.30**	**0.16**	**92.0**	0.4	**0.35**	**0.15**	**87.8**	0.5

Table 1. Comparison of CAD reverse engineering methods on DeepCAD and Fusion360 datasets. Our `CAD-Recode` trained on the 160 k DeepCAD dataset demonstrates an improvement over existing state-of-the-art methods both in terms of geometric fidelity and validity of the generated sketch-extrude sequences. Our procedurally generated dataset provides a significant boost in the prediction quality.

5. Experiments

In order to validate the effectiveness of `CAD-Recode`, we conduct a series of experiments across two different scenarios. The first scenario focuses on the reverse engineering task, where the goal is to reconstruct a CAD sketch-extrude sequence in Python code from a given input point cloud. The second assesses the interpretability and editability of the generated CAD code with a proprietary LLM [36].

5.1. Reverse Engineering

Experimental Setup: `CAD-Recode` is evaluated on three test datasets: DeepCAD [48] (8046 models), Fusion360 [47] (1725 models), and the real-world CC3D [35] (2973 models). The point clouds are obtained by sampling points on the meshes for DeepCAD and Fusion360. The CC3D dataset provides a real-world scenario with input point clouds sampled from actual 3D scans of CAD models containing surface noise, smoothed edges and missing parts (see supplementary materials for more details). Implementation details are provided in the supplementary.

Metrics: To evaluate the quality of the predicted CAD sketch-extrude sequences, we use three metrics: Chamfer Distance (CD) [21], Intersection over Union (IoU) [34], and Invalidity Ratio (IR) [48]. We report both mean and median CD values computed using 8192 points to assess geometric accuracy. Reported CD values have been multiplied by 10^3. The IoU is computed from the resulting CAD model meshes and expressed as a percentage. The IR indicates the percentage of generated sequences that fail to produce a valid CAD model.

Results & Analysis: Table 1 presents results on the test sets of DeepCAD and Fusion360 datasets, where `CAD-Recode` establishes new state-of-the-art performance across all metrics. Note that the results of state-of-the-art methods in Table 1 are borrowed from [34], except for CAD-SIGNet [21], MultiCAD [33], TransCAD [12], and DeepCAD [48] which were taken from [21] and [12]. First, we convert the DeepCAD dataset (160 k models) to CadQuery Python code and use it to train `CAD-Recode` (results are in row before last of Table 1). When trained on DeepCAD dataset as existing methods, `CAD-Recode` outperforms them in almost all metrics. These results showcase the effectiveness of `CAD-Recode` and the proposed CAD code representation.

Training on 1 M generated samples results in substantial improvements in CD and IoU metrics while maintaining a negligibly low invalidity ratio (last row of Table 1), reflecting significantly better geometric fidelity in the predicted CAD models. `CAD-Recode` demonstrates a ten-fold improvement in mean CD and an increase of IoU by over 10% on both DeepCAD and Fusion360 datasets compared to the existing best methods. These results confirm that our large-scale procedurally generated training dataset provides substantial benefits.

As illustrated in Figure 5, this translates to consistent reconstruction quality, where `CAD-Recode` reliably produces CAD models that accurately capture the geometry from the input point cloud. In contrast, CAD-SIGNet [21] can generate shapes that deviate significantly from the target geometry, further highlighting the advantages of our approach.

Real-world Scenario: In Table 2, we evaluate `CAD-Recode` on the real-world CC3D dataset, where input point clouds are sampled from 3D scans and contain artifacts such as surface noise, smooth edges, and missing parts. Even under these challenging conditions, our method achieves significant improvements over CAD-SIGNet [21], with a 89% lower median CD and a 30% higher IoU, while maintaining a low IR. From the CC3D qualitative results in Figure 5, `CAD-Recode` is able to recover

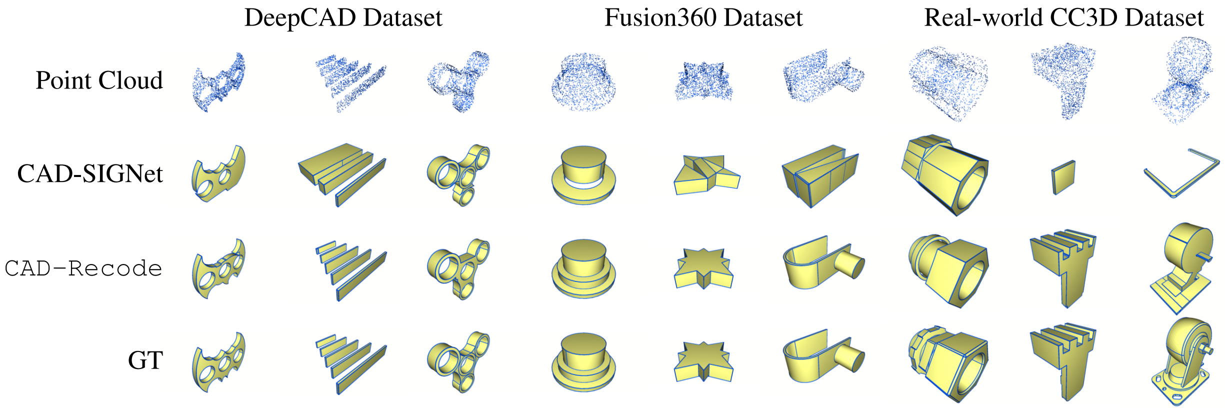


Figure 5. Qualitative results on the DeepCAD, Fusion360, and CC3D datasets. For each input point cloud (first row), we compare CAD models produced by CAD-SIGNet (second) and our `CAD-Recode` trained on our dataset (third) with a ground truth CAD model (bottom row). While CAD-SIGNet often fails to restore the general shape, CAD-Recode outputs only slightly deviate from ground truth in most cases.

Method	Mean CD↓	Med. CD↓	IoU↑	IR↓
DeepCAD [48]	–	263	–	12.7
CAD-SIGNet [21]	14.82	2.90	42.6	2.5
`CAD-Recode`	**0.76**	**0.31**	**74.2**	**0.3**

Table 2. Results on the CC3D dataset, where input point clouds are sampled from real 3D scans. `CAD-Recode` significantly outperforms DeepCAD, and CAD-SIGNet.

geometries that are much closer to the ground truth than current state-of-the-art. However, it can be observed that `CAD-Recode` still lacks the expressiveness to model complex shapes that contain operations beyond the extrusion operation such as revolution and fillet. This can be attributed to the choice of features and design practices in the procedurally generated training dataset. Nevertheless, we believe that this can be addressed in future works by incorporating further features in the dataset generation procedure. Our results on CC3D are compared with methods previously reported for this dataset [21], namely CAD-SIGNet and DeepCAD.

Ablation Study: To evaluate the different components of `CAD-Recode`, we conduct a comprehensive ablation study on the amount of training data, test-time sampling, and the number of input points and model parameters.

Training `CAD-Recode` on 160 k procedurally generated samples using the method described in Section 3.2 leads to significant improvements in geometric fidelity of the predicted samples over training on the DeepCAD dataset with the same amount of data (see row 2 and 3 of Table 3). Furthermore, scaling our training dataset to 1 M samples provides further improvements across all datasets (row 4 of Table 3). As compared to DeepCAD training dataset, our procedural dataset generation provides a better way of learning the mapping between point clouds and CAD codes which can be further improved by scaling up the dataset size.

We investigate the effectiveness of the test-time sampling approach that generates multiple CAD code predictions through different point cloud samplings, as described in Section 4.3. As shown in the third and last row of Table 3, the test-time sampling approach mainly helps reducing the ratio of invalid predicted CAD codes (IR). For comparison, CAD-SIGNet [21] employs a probability-based sampling. Yet, even without test-time sampling our method still performs better on the reconstruction metrics than CAD-SIGNet [21].

Results in Table 4 show an ablation of the input point cloud size and the number of parameters of the LLM backbone. It can be observed that using an input point cloud of 256 points and Qwen1.5b results in the highest IoU. This setting with a relatively small input point cloud and lightweight LLM backbone provides the best balance between prediction accuracy and memory requirements. Results on all metrics are included in the supplementary materials.

5.2. CAD-QA and Editability

CAD-QA and LLM Interpretability: CAD SGP-Bench [38] is a benchmark of 1000 CAD-specific Question Answering (CAD-QA) tasks that test LLMs' understanding of CAD model geometry from sketch-extrude sequences encoded as in DeepCAD [48]. These questions require analyzing various geometric aspects, such as relative sizes and 3D primitive types. We extend this benchmark to evaluate CAD reverse engineering methods by using point clouds as input instead of CAD sequences. Our evaluation follows a two-stage process: first predicting sketch-extrude sequences from point clouds as CadQuery code with `CAD-Recode`, then using GPT-4o [36] to an-

Method	Train Dataset Name	Train Dataset Size	Test-time Sampling	DeepCAD CD↓	DeepCAD IoU↑	DeepCAD IR↓	Fusion360 CD↓	Fusion360 IoU↑	Fusion360 IR↓	Real-World CC3D CD↓	Real-World CC3D IoU↑	Real-World CC3D IR↓
Previous best [21]	DeepCAD	160 k	✓	3.43	77.6	0.9	7.37	65.6	1.6	14.80	42.6	4.4
`CAD-Recode`	DeepCAD	160 k	✓	0.89	86.2	**0.0**	1.77	75.6	**0.0**	2.43	62.5	**0.0**
`CAD-Recode`	Ours	160 k	✓	0.54	88.3	0.3	0.66	82.0	0.1	1.27	69.0	0.2
`CAD-Recode`	Ours	1 M	✓	**0.30**	**92.0**	0.4	**0.35**	**87.8**	0.5	**0.76**	**74.2**	0.3
Previous best [21]	DeepCAD	160 k	✗	6.81	77.3	4.4	14.5	58.4	9.3	32.59	39.1	15.5
`CAD-Recode`	Ours	1 M	✗	**0.83**	**87.1**	**3.1**	**1.21**	**79.1**	**5.0**	**3.21**	**60.5**	**9.8**

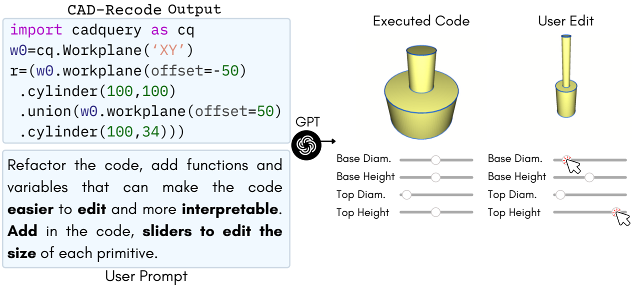
Table 3. Ablation of training data and test-time sampling. The results demonstrate the advantage of training on our procedurally generated data, while the test-time sampling helps reducing the invalidity ratio. CD stands for mean Chamfer distance.

Points	Model Size	DeepCAD CD↓	DeepCAD IoU↑	Fusion360 CD↓	Fusion360 IoU↑	CC3D CD↓	CC3D IoU↑
128	0.5 B	0.18	89.9	0.18	84.3	0.38	71.9
256	0.5 B	0.17	90.6	0.17	85.4	0.36	72.6
256	1.5 B	**0.16**	**92.0**	**0.15**	**87.8**	**0.31**	**74.2**

Table 4. Ablation of architecture details. CD stands for median Chamfer distance.

Method	CAD-QA Accuracy↑
PointLLM [49]	42.3
CAD-SIGNet [21] → GPT-4o	63.2
`CAD-Recode` → GPT-4o	**76.5**

Table 5. Point cloud CAD-specific question answering (CAD-QA) on the SGP-Bench benchmark. Our `CAD-Recode` supplied with a GPT-4o significantly outperforms baseline methods.



```
import cadquery as cq
w0=cq.Workplane('XY')
r=(w0.workplane(offset=-50)
 .cylinder(100,100)
 .union(w0.workplane(offset=50)
 .cylinder(100,34)))
```

Refactor the code, add functions and variables that can make the code **easier** to **edit** and more **interpretable**. **Add** in the code, **sliders to edit the size** of each primitive.

Figure 6. Interactive editing of a CAD model. Given the code output from `CAD-Recode` and a generic prompt, GPT-4o allows automated and interactive editing of the CAD model.

swer CAD-specific questions. Without requiring additional interpretation hints, our approach achieves 76.5% accuracy on this CAD-QA task (Table 5). For comparison, we evaluate two baseline approaches: CAD-SIGNet [21] and PointLLM [49]. When using CAD-SIGNet's output with GPT-4o, even with provided interpretation hints explaining the sequence format, the accuracy reaches only 63.2%. PointLLM, which directly processes point clouds for language tasks, achieves 42.3% accuracy when prompted with the CAD-specific questions. These results demonstrate that `CAD-Recode` effectively captures CAD geometric information while generating an output in a format that proprietary LLMs can naturally interpret and process.

Editing Pipeline: Leveraging the interpretable nature of our code-based output, we present an automated editing pipeline using GPT-4o [36]. Through a simple prompt, the system refactors the generated code to expose geometric parameters via interactive sliders, enabling direct manipulation of the reconstructed model. As shown in Figure 6, the resulting code can be directly executed in a Python environment to provide an interactive editing interface. Implementation details are provided in the supplementary materials.

6. Conclusion

This work rethinks the problem of feature-based CAD reverse engineering by approaching it through pre-trained LLMs taking advantage of CAD Python-based representation. Our key contributions include a new CAD code representation for reverse engineering sketch-extrude sequences, very large-scale procedurally generated training dataset in the form of CAD programs, and a point cloud-to-CAD code model. We demonstrate that `CAD-Recode` outperforms existing methods by a large margin on three datasets, including the real-world CC3D dataset. We also showcase that combining pre-trained LLMs with geometric understanding enables powerful new workflows, where designers can reconstruct CAD models from point clouds and modify them through natural language. We believe that this work will open new perspectives for CAD reverse engineering. We identify the following interesting future works: (1) further exploiting the modularity of the proposed CAD code representation, (2) scaling up the LLM and the dataset to enable reverse engineering of more complex CAD models.

Acknowledgement: The present project is supported by the National Research Fund, Luxembourg under the BRIDGES2021/IS/16849599/FREE-3D and IF/17052459/CASCADES projects, and by Artec 3D.

A. Training Details

The `CAD-Recode` implementation uses Qwen2-1.5B as the LLM decoder. The training configuration employs the AdamW optimizer with a learning rate of 0.0002 and weight decay of 0.01, while maintaining other parameters at their default values from the HuggingFace implementation [44], including the cosine learning rate scheduler. The training process is conducted for $100\,\text{k}$ iterations, incorporating an initial warmup period of $1\,\text{k}$ iterations. Using a single NVIDIA H100 GPU with a batch size of 18, the complete training process takes approximately 12 hours. For ablation study examining decoder size impact (Section 5.1 of the main paper), we utilize Qwen2-0.5B.

B. Training Dataset Generation Algorithm

In Section 3.2, the procedurally generated training dataset is presented. The main advantage of generating data over using the existing DeepCAD dataset for training is that the algorithm allows full control over the amount of data as well as the features and design patterns that the network is exposed to during training. We generate one million valid Python CadQuery code snippets, through an automated pipeline leveraging PythonOCC [37] and CadQuery [10]. The generation process consists of two primary components: (1) a sketch profile generator (Algorithm 1) that creates valid 2D sketches, and (2) a CAD model generator (Algorithm 2) that produces 3D CAD models from these sketches.

The sketch generation process combines primitive shapes (circles and rectangles) through boolean operations (union and cut). From each generated sketch, we extract the primitives (lines, arcs, and circles) from both inner and outer loops. The validity of the generated sketch is ensured through multiple verification steps, including verifying that loops do not intersect, and each primitive has a length greater than zero. Finally, we ensure that the randomly generated CAD code has not previously been generated using the duplicate detection protocol outlined in [51]. This ensures that each sample in the dataset is unique.

The CAD model generation procedure extrudes the validated sketches and combines them through union operations. The planes on which the sketches lie are randomly generated by choosing one of the three canonical planes translated by a random amount. Each resulting model undergoes normalization to fit within a unit bounding box centered at the origin. The parameters are quantized so that the coordinates of any point on the CAD surface are within the range -100 to 100 with a minimum resolution of 1 unit. We then simplify the sequence using higher level abstractions (rectangle, box, and cylinder) by considering the sequence parameters. Our validation framework verifies that a generated code w executes without errors (ϕ_{syn}). Furthermore, we check that the executed code produces a geometric valid CAD model (ϕ_{cad}) using the `BRepCheck_Analyzer` function from PythonOCC as in [48]. Invalid models are excluded from the dataset.

Figure 7 presents examples of CAD models alongside their corresponding CadQuery Python code from our procedurally generated dataset. It is worth noting that the generated codes are fairly compact, this was designed to facilitate training. All code examples are directly executable using a standard Python interpreter with the CadQuery library. The codes follow a consistent three-part structure: (1) necessary library import, (2) definition of sketch planes, and (3) sketch-extrude operations combined through union.

The training dataset generation procedure provides full control over the features included. In Figure 8, it can be observed that the distribution of our CAD models is skewed towards models with larger face and edge count per model with interquartile ranges. As a result, our procedurally generated dataset provides a larger variety of models.

C. Real-World CC3D Dataset

Results on the real-world CC3D [7, 35] dataset are presented in Table 3 of the main paper. This scenario provides an experimental evaluation in a realistic setting, as the input point clouds are sampled from actual 3D scans of CAD models. Sample models are depicted in Figure 9, where artifacts such as surface noise, smoothed edges, and missing parts can be observed. Furthermore, several models from the CC3D dataset are constructed using a range of operations beyond simple extrusion, including revolution, chamfer, and fillet. Consequently, the real-world CC3D dataset provides a challenging set of inputs that enables robust in-the-wild evaluation of our proposed method.

D. Further Experimental Results

Qualitative Results: Additional qualitative results for the reverse engineering of CAD models from point clouds are presented for DeepCAD (Figure 10), Fusion360 (Figure 11), and real-world CC3D (Figure 12) datasets. As detailed in Section 5.1 of the main paper, `CAD-Recode` consistently generates shapes that closely approximate the input point cloud geometry, whereas CAD-SIGNet [21] can generate predictions that greatly differ from the input.

Code Outputs: Figure 13 illustrates the predicted code sequences and their corresponding reconstructed shapes. The predicted codes have a syntax that is consistent with the procedurally generated training examples, showing that

Algorithm 1 Generate2DSketch

```
function GENERATE2DSKETCH
    $numPrimitives \leftarrow RandInt(3, 8)$    ▷ Choose random number of shape primitives
    $compositeShape \leftarrow \emptyset$    ▷ Initialize empty shape
    for $i \leftarrow 1$ to $numPrimitives$ do    ▷ Build shape by combining primitives
        $primitive \leftarrow$ random from $\{$Circle, RotatedRectangle$\}$
        $booleanOperation \leftarrow$ random from $\{$Union, Cut$\}$    ▷ Union adds, Cut subtracts
        $compositeShape \leftarrow ApplyOperation(compositeShape, primitive, booleanOperation)$
    end for
    $boundaryLoops \leftarrow ExtractBoundaryLoops(compositeShape)$    ▷ Extract shape boundaries
    $boundaryComponents \leftarrow \emptyset$
    for $loop \in boundaryLoops$ do    ▷ Process each boundary loop
        $(edgeSequence, isOuter) \leftarrow AnalyzeBoundary(loop)$    ▷ List of parametric curves (lines, arcs, circles)
        $boundaryComponents.Append((edgeSequence, isOuter))$
    end for
    $boundaryComponents \leftarrow ValidateShapeTopology(boundaryComponents)$    ▷ Ensure valid shape topology
    return $boundaryComponents$    ▷ Returns list of (edges, boolean) tuples
end function
```

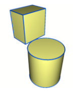
Algorithm 2 GenerateCAD

```
function GENERATECAD
    $cadModel \leftarrow \emptyset$    ▷ Initialize empty CAD model
    $planes \leftarrow GenerateRandomPlanes()$    ▷ Create set of reference planes
    $sketches \leftarrow Generate2DSketch()$    ▷ Get sketches from Algorithm 1
    for $sketch \in sketches$ do    ▷ Create 3D volumes from sketches
        $plane \leftarrow RandomSelect(planes)$    ▷ Select random reference plane
        $volume \leftarrow ExtrudeSketch(sketch, plane)$    ▷ Create 3D volume by extrusion
        $cadModel \leftarrow BooleanUnion(cadModel, volume)$    ▷ Add volume to model
    end for
    $cadModel \leftarrow NormalizeModel(cadModel)$    ▷ Ensure the model fits within a unit box
    $cadModel \leftarrow QuantizeParameters(cadModel)$    ▷ Discretize model parameters
    $cadModel \leftarrow SimplifyCADModel(cadModel)$    ▷ Identify high-level abstractions (rectangle, box, and cylinder)
    $cadModel \leftarrow ValidateCADModel(cadModel)$    ▷ Ensure validity of CadQuery code and CAD model geometry
    $cadModel \leftarrow CheckDuplicate(cadModel)$    ▷ Ensure that the sequence has not previously been generated.
    return $cadModel$
end function
```

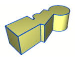
`CAD-Recode` successfully learns both the features and CAD design patterns established in the training set.

Ablation Results: Table 6 shows the architecture ablation results on all metrics, complementing Table 4 of the main paper. Results show that for the same size of input point clouds Qwen1.5b always produces better geometric performance (median CD and IoU) than Qwen0.5b. This can be attributed to the higher number of parameters as well as to the better ability of the model to produce valid python code before fine-tuning. Furthermore, increasing the size of the input point cloud demonstrates a similar pattern, with Qwen1.5b with an 256 input points appears to be the set of architecture parameters leading to the best performance. Note that the mean CD is a metric that is very sensitive to outlier predictions. While Qwen1.5b with 256 input points appears to result in the highest IR, it is negligibly low on all datasets (less 0.5%). This can also be explained by the fact that this setting produces more complex CAD sketch-extrude sequences, making them more susceptible to errors. Note that a key idea of our method is to leverage pre-trained LLMs as decoder of Python code. In the absence of LLM-based CAD reverse engineering methods, we compare our approach to SOTA methods despite the difference in model sizes. For reference, CAD-SIGNet contains 6 M parameters.

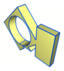
Command & Parameter Accuracy: In order to evaluate the ability of `CAD-Recode` to predict numerical values and sequences that are consistent with the training set, we evaluate `CAD-Recode` trained on the DeepCAD dataset converted to CADQuery python codes with the $\text{Acc}_{\text{command}}$ and $\text{Acc}_{\text{parameter}}$ as introduced in [48]. The results on the DeepCAD testing set are presented in Table 7. It can be observed that `CAD-Recode` achieves comparable performance to the state-of-the-art on the commnad type accuracy and significantly higher performance on the parameter accuracy. This demonstrates that `CAD-Recode` is able to predict numerical values accurately. Note that, those metrics were originally developed to evaluate autoencoding ability. However, there may exist many different possible valid CAD sequences to construct the same CAD model and these metrics do not take this into account. As a result, these metrics were omitted in recent works (CAD-SIGNet [21] and



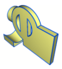
```
import cadquery as cq
w0 = cq.Workplane('ZX', origin=(0, -13, 0))
r = w0.workplane(offset=-87 / 2).moveTo(52.5, 10.5).box(57, 83, 87)
  .union(w0.workplane(offset=23 / 2).moveTo(-29, 0).cylinder(23, 30))
  .union(w0.workplane(offset=113 / 2).moveTo(-29, 0).cylinder(113, 52))
```



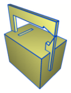
```
import cadquery as cq
w0 = cq.Workplane('ZX', origin=(0, -30, 0))
r = w0.sketch().segment((-30, -27),(-18, -31)).segment((-19, -31)).segment((-19, -100))
    .segment((38, -100)).segment((38, -31)).segment((10, -31)).segment((13, -23))
    .arc((30, -13), (23, 5)).segment((33, 33)).segment((16, 39)).arc((-12, 99),(-9, 33))
    .close().assemble().finalize().extrude(60)
```



```
import cadquery as cq
w0 = cq.Workplane('YZ', origin=(-14, 0, 0))
r = w0.workplane(offset=17 / 2).moveTo(4, -73.5).box(104, 53, 17)
  .union(w0.sketch().segment((-78, 23), (2, -55)).segment((40, -17))
    .arc((42, -24),(48, -30)).segment((48, 5)).segment((61, 5))
    .segment((78, 22)).segment((-2, 100)).close().assemble()
    .push([(0, 22)]).circle(50, mode='s').finalize().extrude(29))
```

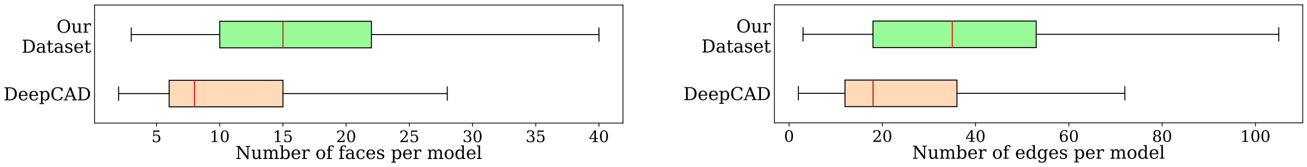


```
import cadquery as cq
w0 = cq.Workplane('XY', origin=(0, 0, 42))
w1 = cq.Workplane('YZ', origin=(-17, 0, 0))
r = w0.sketch().arc((-12, 6), (34, -29),(-1, 16)).segment((5, 4)).segment((-8, -2))
    .close().assemble().finalize().extrude(56)
  .union(w0.sketch().arc((-42, 54), (-12, 71), (19, 54)).segment((19, 78))
    .segment((-42, 78)).close().assemble().finalize().extrude(58))
  .union(w1.sketch().segment((-44, -100), (51, -100)).segment((51, 5)).segment((27, 5))
    .arc((-58, 40),(-44, -51)).close().assemble().reset()
    .face(w1.sketch().arc((-54, -17), (-26, -34),(3, -17)).close()
    .assemble(), mode='s').reset().face(w1.sketch().segment((-54, 14), (3, 14))
    .arc((-26, 31), (-54, 14)).assemble(), mode='s').finalize().extrude(-13))
```



```
import cadquery as cq
w0 = cq.Workplane('YZ', origin=(-22, 0, 0))
w1 = cq.Workplane('ZX', origin=(0, -19, 0))
r = w0.sketch().segment((-100, -83),(-67, -83)).segment((-80, -52)).segment((-75, -50))
    .segment((-75, 62)).segment((17, 62)).segment((17, -62)).segment((-40, -62))
    .segment((-37, -71)).segment((-65, -83)).segment((43, -83))
    .segment((43,83)).segment((-100,83)).close().assemble().finalize().extrude(8)
  .union(w1.sketch().segment((-77, -53),(76, -53)).arc((76, -48),(77, -42))
  .segment((77, 53)).segment((-77, 53)).close().assemble()
  .push([(38.5, 2.5)]).rect(9, 57, mode='s').finalize().extrude(119))
```

Figure 7. Examples from our procedurally generated training dataset. Each row contains CadQuery Python code and a corresponding CAD model. Examples contain not only basic *line*, *circle*, and *arc* primitives, but also higher-level abstractions such as *rect*, *box*, and *cylinder*.



(a) Box-plot graph of the distribution of the number of faces per model.

(b) Box-plot graph of the distribution of the number of edges per model.

Figure 8. Our 1 M procedurally generated training dataset displays distributions CAD models that are skewed towards models with larger edge and face count per model than the DeepCAD dataset (160 k models).

TransCAD [12]).

Invalid Predictions: The invalidity rate of `CAD-Recode` predictions is very low, below 1% on the DeepCAD [48], Fusion360 [47] and real-world CC3D [35] dataset. Some examples of invalid code predictions are presented in Figure 14. Invalid predictions happen when the CAD model contains features of dimension smaller than the resolution induced by quantization (Figure 14(a) and (b)) or when the ground truth CAD model contains features, such as revolution or B-spline, that are not present in the training dataset (Figure 14(c) and (d))).

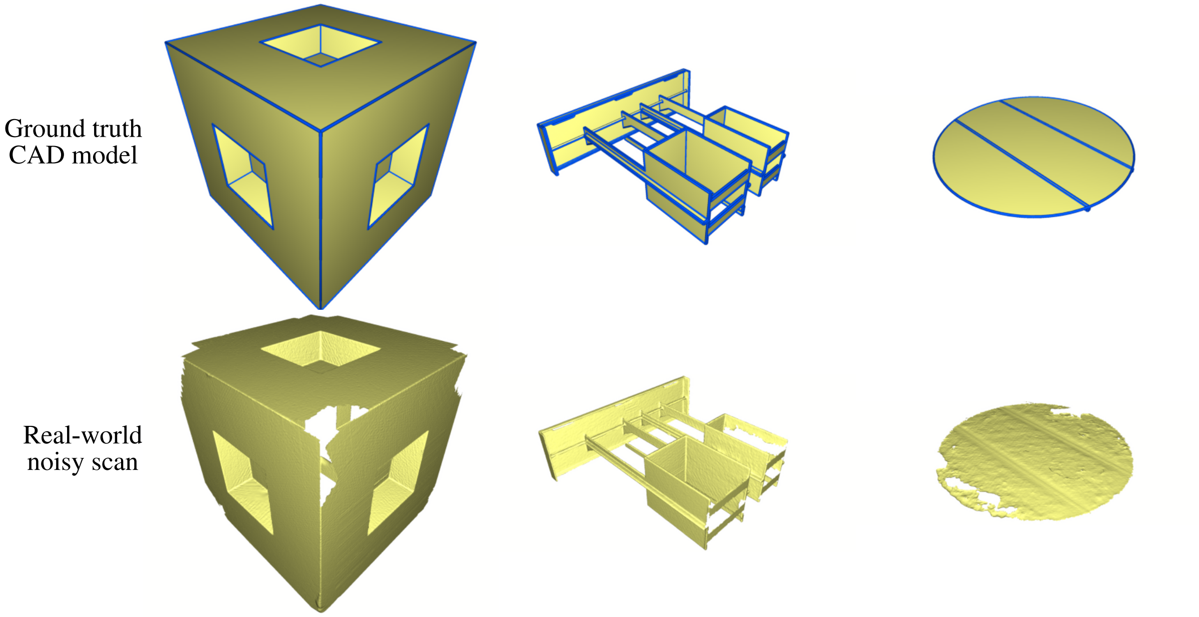


Figure 9. Example models from real-world CC3D dataset. The scans exhibits numerous artifacts such as surface noise, missing parts and smoothed edges. In the CC3D experiments reported in the main paper, the input point clouds are sampled from the scans. Zoom in for better details.

Points	Model Size	DeepCAD				Fusion360				CC3D			
		Mean CD↓	Med. CD↓	IoU↑	IR↓	Mean CD↓	Med. CD↓	IoU↑	IR↓	Mean CD↓	Med. CD↓	IoU↑	IR↓
64	0.5 B	0.42	0.20	88.5	0.1	0.58	0.22	82.1	0.1	0.87	0.45	70.1	0.1
	1.5 B	0.36	0.19	89.3	0.0	0.43	0.20	83.7	0.1	0.83	0.42	71.2	0.0
128	0.5 B	0.36	0.18	89.9	0.1	0.43	0.18	84.3	0.1	0.87	0.38	71.9	0.1
	1.5 B	**0.27**	0.17	91.0	0.1	0.36	0.17	86.1	0.1	0.79	0.34	73.1	0.1
256	0.5 B	0.36	0.17	90.6	0.2	0.40	0.17	85.4	0.4	0.87	0.36	72.6	0.1
	1.5 B	0.30	**0.16**	**92.0**	0.4	**0.35**	**0.15**	**87.8**	0.5	**0.76**	**0.31**	**74.2**	0.3

Table 6. Ablation of architecture details.

Method	$\text{Acc}_{\text{command}}$ (%)	$\text{Acc}_{\text{parameter}}$ (%)
DeepCAD [48]	80.4	69.6
PrismCAD [25]	73.0	66.8
HNC-CAD [52]	82.7	74.6
CAD-Diffuser [34]	88.5	82.9
`CAD-Recode`	83.9	92.1

Table 7. Command and parameter accuracy results [48] on the DeepCAD dataset. All methods (incl. `CAD-Recode`) are trained and tested on DeepCAD dataset.

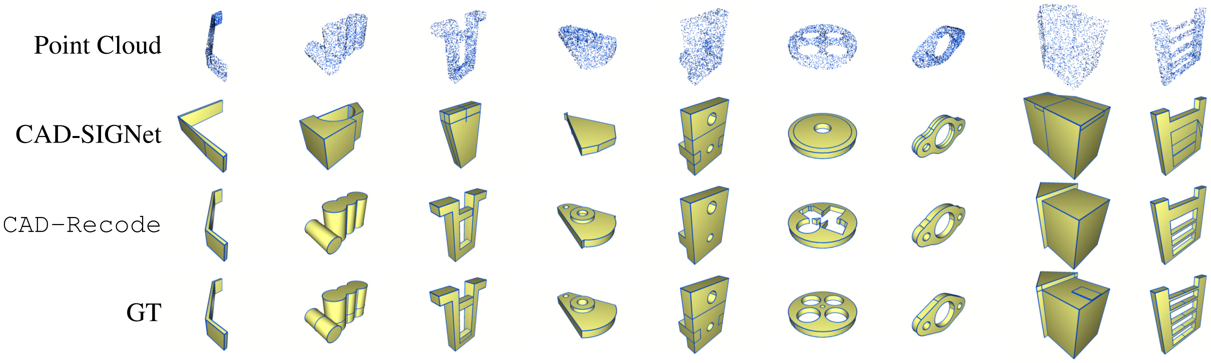


Figure 10. Qualitative results on the DeepCAD dataset.

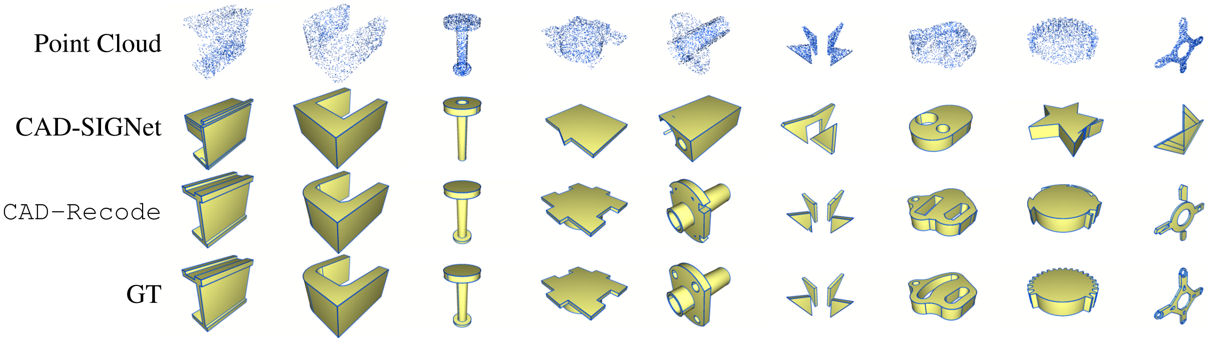


Figure 11. Qualitative results on the Fusion360 dataset.

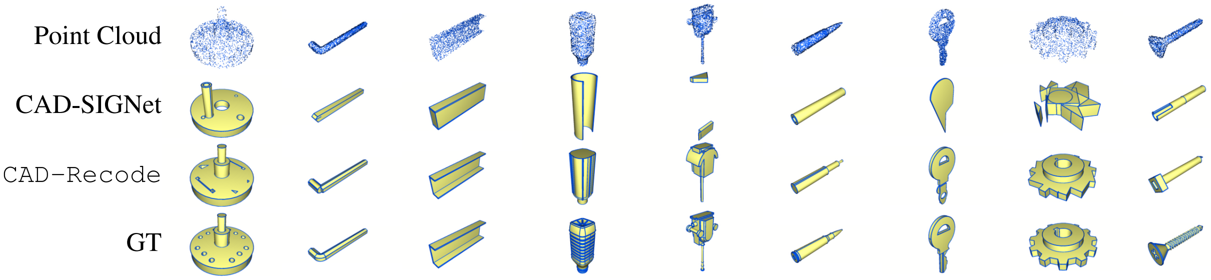
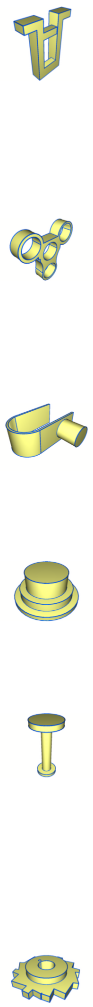


Figure 12. Qualitative results on the real-world CC3D dataset.



```
import cadquery as cq
w0 = cq.Workplane('XY', origin=(0, 0, -17))
r = w0.sketch().segment((-48, -64), (24, -64)).segment((24, -43)).segment((-27, -43))
    .segment((-27, 60)).segment((27, 60)).segment((27, -39)).segment((48, -39))
    .segment((48, 100)).segment((-48, 100)).close().assemble().finalize().extrude(20)
  .union(w0.sketch().segment((-82, -100),(-27, -100)).segment((-27, 80)).segment((27, 80))
    .segment((27, -100)).segment((82, -100)).segment((82, -79)).segment((48, -79))
    .segment((48, 100)).segment((-48, 100)).segment((-48, -79)).segment((-82, -79))
    .close().assemble().finalize().extrude(34))
```

```
import cadquery as cq
w0 = cq.Workplane('XY', origin=(0, 0, -16))
r = w0.sketch().arc((-46, -23), (-95, -74), (-27, -56)).segment((30, -56))
    .arc((96, -72), (44, -25)).segment((44, -12)).arc((31, 14), (30, 42))
    .arc((1, 92), (-31, 44)).arc((-32, 43), (-33, 43)).arc((-31, 20),(-39, -2))
    .segment((-39, -12)).segment((-43, -12)).arc((-45, -17),(-46, -23)).assemble()
    .push([(-64, -56)]).circle(28, mode='s').push([(0, 56)]).circle(28, mode='s')
    .push([(0, -19)]).circle(28, mode='s').push([(65, -56)])
    .circle(28, mode='s').finalize().extrude(32)
```

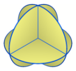
```
import cadquery as cq
w0 = cq.Workplane('ZX', origin=(0, 40, 0))
w1 = cq.Workplane('XY', origin=(0, 0, -19))
r = w0.sketch().arc((-24, -47), (41, -99),(87, -32)).segment((88, -32)).segment((88, 100))
    .segment((82, 100)).segment((82, -52)).arc((34, -94), (-18, -52)).segment((-18, 100))
    .segment((-24, 100)).close().assemble().finalize().extrude(-80)
  .union(w1.workplane(offset=-69 / 2).moveTo(52, 0).cylinder(69, 32))
```

```
import cadquery as cq
w0 = cq.Workplane('ZX', origin=(0, 31, 0))
r = w0.workplane(offset=-75 / 2).cylinder(75, 62)
  .union(w0.workplane(offset=-25 / 2).cylinder(25, 81))
  .union(w0.workplane(offset=13 / 2).cylinder(13, 100))
```

```
import cadquery as cq
w0 = cq.Workplane('ZX', origin=(0, 69, 0))
w1 = cq.Workplane('ZX', origin=(0, -85, 0))
r = w0.workplane(offset=-150 / 2).cylinder(150, 15)
  .union(w0.workplane(offset=10 / 2).cylinder(10, 31))
  .union(w0.workplane(offset=31 / 2).cylinder(31, 8))
  .union(w1.workplane(offset=-15/2).cylinder(15,46))
```

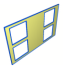
```
import cadquery as cq
w0 = cq.Workplane('ZX', origin=(0, 20, 0))
r = w0.sketch().circle(61).circle(25, mode='s').push([(34, 4)])
    .circle(4, mode='s').finalize().extrude(-41)
  .union(w0.sketch().segment((-100, 19), (-88, 11)).segment((-97, -34)).segment((-67, -41))
    .segment((-77, -66)).segment((-57, -74)).segment((-57, -72)).segment((-56, -72))
    .segment((-56, -75)).segment((-32, -80)).segment((-35, -95)).segment((-16, -100))
    .segment((-11, -83)).segment((33, -100)).segment((45, -70)).segment((68, -76))
    .segment((76, -61)).segment((66, -56)).segment((100, -30)).segment((88, -19))
    .segment((97, 34)).segment((67, 41)).segment((77, 66)).segment((57, 74))
    .segment((51, 69)).segment((51, 70)).segment((32, 76)).segment((35, 95))
    .segment((16, 100)).segment((11, 83)).segment((-33, 100)).segment((-45, 70))
    .segment((-68, 77)).segment((-76, 62)).segment((-66, 56)).close().assemble()
    .circle(26, mode='s').finalize().extrude(-20))
```

Figure 13. `CAD-Recode` predictions on DeepCAD (top 2 rows), Fusion360 (mid 3 rows), and CC3D (last row) datasets. Each row contains predicted CadQuery Python code and its result after execution in Python interpreter.



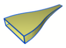
```python
import cadquery as cq
w0 = cq.Workplane('XY', origin=(0, 0, 0))
w1 = cq.Workplane('YZ', origin=(0, 0, 0))
r = w0.workplane(offset=0 / 2).cylinder(0, 98)
  .union(w1.workplane(offset=0 / 2).cylinder(0, 100))
```

(a) The ground truth model contains three very thin cylinders with height smaller than 1. As a result, `CAD-Recode` is not able to predict heights with sufficient precision due to quantization and predicts cylinders with height 0, producing an invalid model.



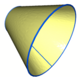
```python
import cadquery as cq
w0 = cq.Workplane('XY', origin=(0, 0, 0))
r = w0.sketch().rect(200, 124).push([(-63.5, 25)]).rect(51, 60, mode='s')
    .push([(55, -25)]).rect(50, 60, mode='s').finalize().extrude(0)
```

(b) As the ground-truth model has thickness less than 1, `CAD-Recode` predicts an extrusion distance of 0 as a quantized approximation (highlighted in yellow), resulting in an invalid CAD model.



```python
import cadquery as cq
w0 = cq.Workplane('YZ', origin=(34, 0, 0))
w1 = cq.Workplane('XY', origin=(0, 0, 44))
r = w0.sketch().segment((-7, -35), (11, -36)).segment((11, -24)).arc((1, -14),
    (6, -2)).segment((-1, 19)).segment((11, 23)).segment((11, 28))
    .segment((11, 29)).segment((12, 29)).segment((12, 35))
    .segment((-4, 36)).close().assemble().finalize().extrude(-133)
  .union(w0.sketch().segment((5, -7), (14, -2)).segment((8, 8)).arc((7, 0),
    (5, -7)).assemble().finalize().extrude(63))
  .union(w1.sketch().arc((-100, 12), (-85, 10), (-70, 5)).arc((-68, 6),
    (-66, 5)).arc((-59, 4), (-52, 2)).arc((-51, 3), (-50, 4)).arc((-72, 7),
    (-90, 12)).close().assemble().finalize().extrude(-88))
```

(c) The ground-truth CAD model is created with B-spline primitives. Since `CAD-Recode` supports only arc, circle and line primitives, it tries to approximate the solution with multiple arcs, but fails to provide a valid CAD model. In particular, the prediction contains an arc constructed from three co-linear points (highlighted in yellow), which raises an error in CadQuery.



```python
import cadquery as cq
w0 = cq.Workplane('XY', origin=(0, 0, -79))
r = w0.sketch().segment((-100, -1), (-91, -1)).arc((0, -93),(91, -1))
    .segment((100, -1)).segment((100, 1)).segment((91, 1)).arc((0, 99),
    (-91, 1)).segment((-100, 1)).close().assemble().push([(0, -2)])
    .circle(90, mode='s').finalize().extrude(-2)
  .union(w0.workplane(offset=140 / 2).cylinder(140, 72))
  .union(w0.sketch().segment((-51, 15), (-50, 15)).arc((0, -53),(50, 15))
          .segment((51, 15)).segment((51, 27)).segment((48, 27)).arc((0, -53),
          (-48, 27)).segment((-51, 27)).close().assemble().finalize().extrude(159))
```

(d) The ground-truth CAD model is created with a revolution operation. Since `CAD-Recode` supports only extrusion operation, it tries to approximate the solution with multiple arcs. However, one of the sketch (highlighted in yellow) results in a self-intersecting loop, which is not a valid face.

Figure 14. Examples of invalid predictions. Each row contains the ground-truth CAD model (left) and an invalid predicted CadQuery Python code (right). The CAD models in (a) and (b) are taken from the DeepCAD dataset and the CC3D dataset for (c) and (d). Invalid predictions mostly take place when the ground-truth contains features of very small dimension with respect to the size of the CAD model as in (a) and (b), or when the ground-truth model contains operations other than the ones supported as in (c) and (d).

E. Test-time Sampling

The ablation study in Section 5.1 of the main paper demonstrates the effectiveness of our test-time sampling strategy. This approach generates multiple plausible solutions by sampling different input point clouds. Figure 15 illustrates the qualitative results from different sampling instances. While `CAD-Recode` successfully captures the overall geometry across different samplings, fine-grained details may vary in reconstruction quality due to the relatively sparse point cloud input. However, this limitation can be effectively addressed by leveraging multiple sampling iterations to capture different aspects of the input geometry.

F. Interpretability and CAD-QA

In this section, we provide further details on the CAD-QA experiments reported in Section 5.2 of the main paper. We start by providing more details on the SGP-Bench benchmark [38]. Then, we present results further results and examples of GPT-4o outputs.

F.1. Representation and CAD-QA

The goal of the SGP-Bench benchmark is to evaluate the spatial-semantic reasoning skills of LLMs from symbolic graphics programs [38]. One aspect of the benchmark is a set of 1000 multiple choice questions on 3D CAD models given their corresponding sketch-extrude sequence in the DeepCAD [48] format. An example is depicted in Figure 16.

To evaluate the interpretability of our code-based CAD representation, we translated the 1000 questions of SGP-Bench from the DeepCAD representation (Figure 16(a)) to the CadQuery code format (Figure 16(b)). Using the same protocol as in SGP-Bench [38], and GPT-4o [36], we found that the accuracy on the multiple choice question in CadQuery format is 82.4%. This is about 4% higher than using the DeepCAD format with an interpretative hint. This suggests the proposed code representation provides a more structured and naturally LLM-interpretable representation of CAD models.

F.2. Point Cloud and CAD-QA

In Table 6 of the main paper, the results for point cloud CAD-QA are presented. Figure 17(a) depicts an example of point cloud and question that was used to obtain these results. In this particular question, the task is to deduce the number of holes present in the CAD model given the point cloud as input. Figure 17(b), the answer provided by PointLLM is shown and it can be observed that PointLLM is unable to retrieve the correct answer. It is worth noting that PointLLM is a network trained to answer semantic questions about object given its point cloud representation, as result in most cases the network is unable to describe geometric CAD-specific questions. For both CAD-SIGNet and `CAD-Recode`, the point cloud CAD-QA is done in a two step process. First the sketch-extrude is sequence is predicted from each network, then the sequence along with the question is passed through GPT-4o. Note that for CAD-SIGNet an interpretative hint is provided to provide context on the structure of the sequence. A sample output for CAD-SIGNet and GPT-4o can be found in Figure 17(c), and in Figure 17(d) for `CAD-Recode` and GPT4-o. As the sequence was incorrectly predicted by CAD-SIGNet the answer to the question is wrong (1 hole), whereas the prediction from `CAD-Recode` captured better the geometry of the input point cloud leading to a correct answer. It is worth noting, that despite not being provided any information about CadQuery Python code in the prompt, GPT-4o is able to breakdown the predicted sequence into its primitive components and provide correct and accurate geometric descriptions. This can be explained by the fact that LLMs are exposed to large amounts of code data during training. As a result, the CadQuery Python representation of CAD models is appropriate for

G. Editing Pipeline Details

We provide more details on the editing pipeline presented in Section 5.2 of the main paper. The goal of this pipeline is to integrate automated editability capabilities to `CAD-Recode`. To this end, we present a simple process using an off-the-shelf LLM, GPT-4o [36]. Starting from an output CAD Python code from `CAD-Recode` as shown in Figure 18a, we prepare a simple and generic prompt (Figure 18b) for the LLM to generate a refactored version of the code such that when executed the user can change with the dimensions of each primitive. As seen in Figure 18c, the LLM is able to generate a code with comments that describe the different primitives semanticallly and include appropriate variables for the dimensions of each of the primitive, such as the height and the diameter of each cylinder. The code generated by the LLM, can be directly executed in a Jupyter notebook with the CadQuery and ipywidgets libraries. Figure 6 of the main paper shows the generated sliders and how can the shape be then edited. This demonstrates that the CAD representation as Python code within a reverse engineering scenario opens the door to new applications when combined with LLMs.

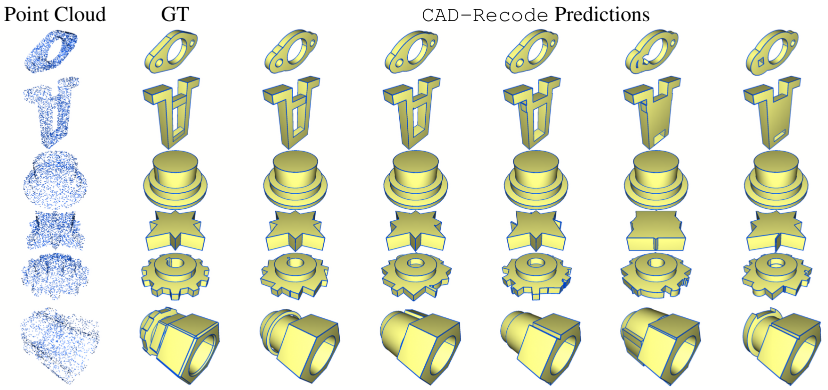


Figure 15. `CAD-Recode` predictions from different point cloud sampling on DeepCAD, Fusion360, and real-world CC3D datasets. For each prediction, 256 points are sampled randomly from the input point cloud.

Examine the following CAD code carefully to understand the 3D object it generates and answer the question based on your interpretation of the rendered image of that object.

```
SOL; Line:(221,128); Line:(221,223) ;Line:(128,223); Line:(128,128); Ext: (128,128,128,32,110,128,98,167,128, Newbody, One-sided); EOS
```

Hint: the CAD code has the following syntax: CAD code consists of a sequence of CAD commands that describe a 3D object. The commands fall into two categories: sketch and extrusion. Sketch commands are used to specify closed curves on a 2D plane in 3D space. Each closed curve is referred as a loop, and one or more loops form a closed region called a profile. A loop always starts with an indicator command <SOL> followed by a series of curve commands. All the curves on the loop are in counterclockwise order, beginning with the curve whose starting point is at the most bottom-left. In total, there are three possible curve commands: Line, Arc, and Circle. Line(x, y): a line, with x, y as line end-point. Arc(x, y, u, f): an arc, with x,y as arc end-point, u as sweep angle and f as whether it is counter-clockwise, f=0 means it is counter-clockwise, f=1 means it is not counter-clockwise. Circle(x, y, r): a circle, with x,y as the center point and r as the radius. The extrusion command has 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 Ext) 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. Ext(x, y, z, o, p, q, s, e, f, b, u): extrude operation, with x, y, z as the sketch plane orientation, o, p, q as the sketch plane origin, s as the scale of the associated sketch profile, e, f as the extrude distances towards both sides, b as the type of merge operation (could be New-body operation, join operation, cut operation and intersect operation) and u as the extrude type (could be one-sided, symmetric or two-sided). <EOS> means the end of the code.

Question: How many faces does the CAD object in the image have?

(a) DeepCAD Representation

Examine the following CAD code carefully to understand the 3D object it generates and answer the question based on your interpretation of the rendered image of that object.

```
import cadquery as cq
def make_shape():
    plane0 = cq.Plane(origin = (-0.75,-0.1406,0.0),xDir = (1.0,0.0,0.0),normal = (0.0,0.0,1.0))
    w0 = cq.Workplane(plane0)
    face0 = w0.sketch().face(w0.sketch().segment( (0.0, 0.0), (0.7495, 0.0)).segment((0.7495, 0.0), (0.7495, 0.7656)).segment((0.7495, 0.7656), (0.0, 0.7656)).segment((0.0, 0.7656), (0.0, 0.0)).assemble(), mode = ’a’).finalize()
    shape0 = face0.extrude(0.3046875, both=False)
    return shape0
```

Question: How many faces does the CAD object in the image have?

(b) CadQuery Representation

Figure 16. Example of a CAD question from SGP-Bench in the DeepCAD representation (b) and using the CadQuery representation (c). When asking the question in CadQuery format no hint is provided, while in DeepCAD format a long interpretative hint is required. The choice of answers are A) 6, B) 4, C) 8 and D) 5. The correct answer is A) 6.

Question: How many holes does the CAD object in the image have?
A) Two, B) Three, C) Four, D) Five
Correct Answer: B

(a) Point cloud and corresponding CAD-specific question

The CAD object in the image has five holes.

(b) PointLLM output from point cloud

To determine the number of holes in the CAD object, let's break down the CAD code step by step:
1. **Extrusion Command:**
Start;Ext:(128,130,128,128,130,128,128,128,NewBody,96);
This command sets up an extrusion with specific parameters and indicates that a new body will be created.
2. **First Loop:**
Line:(43,0);Arc:(213,0,255,43);Line:(213,85);Arc:(43,85,0,43);EOL;
- Line:(43,0) - A line from the starting point to (43,0).
- Arc:(213,0,255,43) - An arc with midpoint (213,0) and endpoint (255,43).
- Line:(213,85) - A line from the previous endpoint to (213,85).
- Arc:(43,85,0,43) - An arc with midpoint (43,85) and endpoint (0,43).
- EOL - End of the loop.

This forms a closed loop, which is a profile for extrusion.
3. **Second Loop:**
Circle:(43,43,43,51);EOL;
- Circle:(43,43,43,51) - A circle with the most left point at (43,43) and the most top point at (43,51).
- EOL - End of the loop.

This forms another closed loop, which is another profile for extrusion.
4. **End of Sketch and Extrusion:**
EOF;EOS;End;
- EOF - End of face.
- EOS - End of sketch.
- End - End of the code.

From the above analysis, we can see that the CAD object consists of:
- One main profile formed by the first loop.
- One circular profile formed by the second loop.

The circular profile represents a hole within the main profile. Since there is only one circular profile, the CAD object has one hole.
Answer: A

(c) GPT-4o output from CAD-SIGNet prediction

Let's break down the CAD code step by step to understand the 3D object it generates and determine the number of holes.
1. **Workplane Creation:**
w0 = cq.Workplane('XY', origin=(0,0,1))
This line creates a workplane on the XY plane with an origin at (0,0,1).
2. **Sketching the Shape:**
r = w0.sketch()
This initializes a sketch on the workplane.
3. **Defining the Shape:**
.segment((-66,-34),(50,-34)) .segment((50,-32)) .arc((100,0),(50,32)) .segment((50,34)) .segment((-66,34)) .segment((-66,32)) .arc((-100,0),(-66,-32)) .close()
These lines define a closed shape with segments and arcs. The shape appears to be symmetrical along the Y-axis.
4. **Adding Holes:**
.assemble() .push([(-66,0)]).circle(10, mode='s') .push([(0,0)]).circle(10, mode='s') .push([(66,0)]).circle(10, mode='s') .finalize()
These lines add three circles (holes) to the shape at positions (-66,0), (0,0), and (66,0).
5. **Extruding the Shape:**
.extrude(-4)
This line extrudes the shape by -4 units, creating a 3D object with the defined shape and holes.
From the code, we can see that three circles (holes) are added to the shape. Therefore, the CAD object has three holes.
Answer: B

(d) GPT-4o output from `CAD-Recode` prediction

Figure 17. Example of Point cloud CAD-QA (a) and answers provided by PointLLM (b), CADSIGNet and GPT-4o (c) and `CAD-Recode` and GPT-4o (d).

```python
import cadquery as cq
w0 = cq.Workplane('XY', origin=(0, 0, 0))
r = w0.workplane(offset=-100 / 2).cylinder(100, 100)
  .union(w0.workplane(offset=100 / 2).cylinder(100, 34))
```

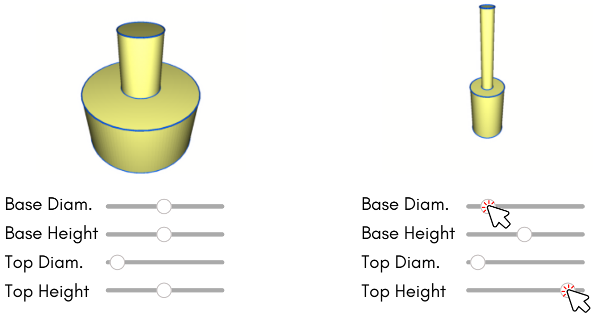
(a) `CAD-Recode` output as CadQuery Python code.

> Refactor the code, add functions and variables that can make the code easier to edit and more interpretable. Add in the code, sliders to edit the size of each primitive: import cadquery as cq w0=cq.Workplane('XY',origin=(0,0,0)) r=w0.workplane(offset=-100/2).cylinder(100,100).union(w0.workplane(offset=100/2).cylinder(100,34))

(b) Prompt for GPT-4o

```python
import cadquery as cq
from ipywidgets import interact, FloatSlider

# Default parameters for the CAD model
default_base_diameter = 100
default_base_height = 100
default_top_diameter = 34
default_top_height = 100

def create_cylinder(workplane, diameter, height, offset):
    """Creates a cylinder on the specified workplane with a given diameter, height, and offset."""
    return workplane.workplane(offset=offset).cylinder(height, diameter / 2)

def build_model(base_diameter=default_base_diameter, base_height=default_base_height,
                top_diameter=default_top_diameter, top_height=default_top_height):
    """Builds the model with adjustable base and top cylinder dimensions."""
    # Initialize the main workplane
    w0 = cq.Workplane("XY", origin=(0, 0, 0))

    # Create base and top cylinders
    base_cylinder = create_cylinder(w0, base_diameter, base_height, offset=-base_height / 2)
    top_cylinder = create_cylinder(w0, top_diameter, top_height, offset=base_height / 2)

    # Union the cylinders
    model = base_cylinder.union(top_cylinder)
    return model

# Function to display the model interactively with sliders for each cylinder dimension
def display_model_with_sliders():
    """Interactive display function with sliders for base and top cylinder dimensions."""

    # Sliders for base and top cylinder dimensions
    base_diameter_slider = FloatSlider(value=default_base_diameter, min=10, max=200, step=5, description="Base Diameter")
    base_height_slider = FloatSlider(value=default_base_height, min=10, max=200, step=5, description="Base Height")
    top_diameter_slider = FloatSlider(value=default_top_diameter, min=10, max=200, step=5, description="Top Diameter")
    top_height_slider = FloatSlider(value=default_top_height, min=10, max=200, step=5, description="Top Height")

    # Update function to apply slider changes
    @interact(
        base_diameter=base_diameter_slider,
        base_height=base_height_slider,
        top_diameter=top_diameter_slider,
        top_height=top_height_slider
    )
    def update_model(base_diameter, base_height, top_diameter, top_height):
        # Build and display the model
        model = build_model(base_diameter, base_height, top_diameter, top_height)
        show_object(model)  # Use this in CQ-editor or a Jupyter environment

# Execute the function to show sliders and interactive model
display_model_with_sliders()
```

(c) GPT-4o output: executable refactored code with interactive sliders



(d) Executed code produces interactive sliders that the designer can use to modify the size of the primitives.

Figure 18. Editing pipeline: given a predicted code from `CAD-Recode` from a point cloud (a), a generic prompt can be constructed to refactor the predicted code to enhance editibility (b). The output from GPT-4o-2024-08-06 is shown in (c), and the generated sliders and possible CAD edits are depicted in (d).

References

[1] Kamel Alrashedy, Pradyumna Tambwekar, Zulfiqar Haider Zaidi, Megan Langwasser, Wei Xu, and Matthew Gombolay. Generating CAD code with vision-language models for 3d designs. In *ICLR*, 2025. 2

[2] Akshay Badagabettu, Sai Sravan Yarlagadda, and Amir Barati Farimani. Query2cad: Generating cad models using natural language queries. *arXiv preprint arXiv:2406.00144*, 2024. 2

[3] Antoine Brière-Côté, Louis Rivest, and Roland Maranzana. Comparing 3d cad models: uses, methods, tools and perspectives. *Computer-Aided Design and Applications*, 9(6): 771–794, 2012. 1

[4] Jorge D Camba, Manuel Contero, and Pedro Company. Parametric cad modeling: An analysis of strategies for design reusability. *Computer-Aided Design*, 74:18–31, 2016. 1

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. 4

[6] Sijin Chen, Xin Chen, Chi Zhang, Mingsheng Li, Gang Yu, Hao Fei, Hongyuan Zhu, Jiayuan Fan, and Tao Chen. Ll3da: Visual interactive instruction tuning for omni-3d understanding reasoning and planning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 26428–26438, 2024. 2

[7] Kseniya Cherenkova, Djamila Aouada, and Gleb Gusev. Pvdeconv: Point-voxel deconvolution for autoencoding cad construction in 3d. In *ICIP*, pages 2741–2745, 2020. 2, 4, 9

[8] Kseniya Cherenkova, Elona Dupont, Anis Kacem, Ilya Arzhannikov, Gleb Gusev, and Djamila Aouada. Sepicnet: Sharp edges recovery by parametric inference of curves in 3d shapes. In *CVPRW*, pages 2726–2734, 2023. 3

[9] Kseniya Cherenkova, Elona Dupont, Anis Kacem, Gleb Gusev, and Djamila Aouada. Spelsnet: Surface primitive elements segmentation by b-rep graph structure supervision. *NeurIPS*, 2024. 3

[10] CADQuery Developers. Cadquery: A python parametric cad scripting framework. `https://cadquery.readthedocs.io/`, 2024. Accessed: 2024-10-22. 2, 3, 4, 9

[11] Elona Dupont, Kseniya Cherenkova, Anis Kacem, Sk Aziz Ali, Ilya Aryhannikov, Gleb Gusev, and Djamila Aouada. Cadops-net: Jointly learning cad operation types and steps from boundary-representations. *3DV*, 2022. 3

[12] Elona Dupont, Kseniya Cherenkova, Dimitrios Mallis, Gleb Gusev, Anis Kacem, and Djamila Aouada. Transcad: A hierarchical transformer for cad sequence inference from point clouds. In *Eur. Conf. Comput. Vis.*, 2024. 2, 3, 4, 6, 11

[13] David W Eggert, Andrew W Fitzgibbon, and Robert B Fisher. Simultaneous registration of multiple range views for use in reverse engineering of cad models. *Computer Vision and Image Understanding*, 69(3):253–272, 1998. 1

[14] Markus Friedrich, Pierre-Alain Fayolle, Thomas Gabor, and Claudia Linnhoff-Popien. Optimizing evolutionary csg tree extraction. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1183–1191, 2019. 3

[15] Haoxiang Guo, Shilin Liu, Hao Pan, Yang Liu, Xin Tong, and Baining Guo. Complexgen: Cad reconstruction by b-rep chain complex generation. *ACM TOG*, 41(4):1–18, 2022. 3

[16] Jiaming Han, Kaixiong Gong, Yiyuan Zhang, Jiaqi Wang, Kaipeng Zhang, Dahua Lin, Yu Qiao, Peng Gao, and Xiangyu Yue. Onellm: One framework to align all modalities with language. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 26584–26595, 2024. 2

[17] Robin H Helle and Hirpa G Lemu. A case study on use of 3d scanning for reverse engineering and quality control. *Materials Today: Proceedings*, 45:5255–5262, 2021. 1

[18] Yining Hong, Haoyu Zhen, Peihao Chen, Shuhong Zheng, Yilun Du, Zhenfang Chen, and Chuang Gan. 3d-llm: Injecting the 3d world into large language models. *NeurIPS*, 2023. 2

[19] Yining Hong, Haoyu Zhen, Peihao Chen, Shuhong Zheng, Yilun Du, Zhenfang Chen, and Chuang Gan. 3d-llm: Injecting the 3d world into large language models. *Advances in Neural Information Processing Systems*, 36:20482–20494, 2023. 2

[20] Kacper Kania, Maciej Zieba, and Tomasz Kajdanowicz. Ucsg-net-unsupervised discovering of constructive solid geometry tree. *Adv. Neural Inform. Process. Syst.*, 2020. 3

[21] Mohammad Sadil Khan, Elona Dupont, Sk Aziz Ali, Kseniya Cherenkova, Anis Kacem, and Djamila Aouada. Cad-signet: Cad language inference from point clouds using layer-wise sketch instance guided attention. In *IEEE Conf. Comput. Vis. Pattern Recog.*, pages 4713–4722, 2024. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

[22] 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 *IEEE Conf. Comput. Vis. Pattern Recog.*, pages 9601–9611, 2019. 2, 4

[23] Milin Kodnongbua, Benjamin Jones, Maaz Bin Safeer Ahmad, Vladimir Kim, and Adriana Schulz. Reparamcad: Zero-shot cad re-parameterization for interactive manipulation. In *SIGGRAPH Asia 2023 Conference Papers*, New York, NY, USA, 2023. Association for Computing Machinery. 2

[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 *IEEE Conf. Comput. Vis. Pattern Recog.*, pages 12773–12782, 2021. 3, 5

[25] Joseph George Lambourne, Karl Willis, Pradeep Kumar Jayaraman, Longfei Zhang, Aditya Sanghi, and Kamal Rahimi Malekshan. Reconstructing editable prismatic cad from rounded voxel models. In *SIGGRAPH Asia*, pages 1–9, 2022. 3, 6, 12

[26] Lingxiao Li, Minhyuk Sung, Anastasia Dubrovina, Li Yi, and Leonidas J Guibas. Supervised fitting of geometric primitives to 3d point clouds. In *IEEE Conf. Comput. Vis. Pattern Recog.*, pages 2652–2660, 2019. 3

[27] Ming Li, Frank C Langbein, and Ralph R Martin. Detecting design intent in approximate cad models using symmetry. *Computer-Aided Design*, 42(3):183–201, 2010. 1

[28] Pu Li, Jianwei Guo, Xiaopeng Zhang, and Dong-Ming Yan. Secad-net: Self-supervised cad reconstruction by learning sketch-extrude operations. In *IEEE Conf. Comput. Vis. Pattern Recog.*, pages 16816–16826, 2023. 2, 3

[29] Pu Li, Jianwei Guo, Huibin Li, Bedrich Benes, and Dong-Ming Yan. Sfmcad: Unsupervised cad reconstruction by learning sketch-based feature modeling operations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4671–4680, 2024. 3

[30] Fei Liu. Fast industrial product design method and its application based on 3d cad system. *Computer-Aided Design and Applications*, 18(S3):118–128, 2020. 1

[31] Minghua Liu, Ruoxi Shi, Kaiming Kuang, Yinhao Zhu, Xuanlin Li, Shizhong Han, Hong Cai, Fatih Porikli, and Hao Su. Openshape: Scaling up 3d shape representation towards open-world understanding. *Advances in neural information processing systems*, 36, 2024. 2

[32] Yujia Liu, Stefano D'Aronco, Konrad Schindler, and Jan Dirk Wegner. Pc2wf: 3d wireframe reconstruction from raw point clouds. *ICLR*, 2021. 3

[33] Weijian Ma, Minyang Xu, Xueyang Li, and Xiangdong Zhou. Multicad: Contrastive representation learning for multi-modal 3d computer-aided design models. In *CIKM*, New York, NY, USA, 2023. Association for Computing Machinery. 2, 3, 4, 6

[34] Weijian Ma, Shuaiqi Chen, Yunzhong Lou, Xueyang Li, and Xiangdong Zhou. Draw step by step: Reconstructing cad construction sequences from point clouds via multimodal diffusion. In *IEEE Conf. Comput. Vis. Pattern Recog.*, pages 27154–27163, 2024. 1, 2, 3, 4, 6, 12

[35] Dimitrios Mallis, Ali Sk Aziz, Elona Dupont, Kseniya Cherenkova, Ahmet Serdar Karadeniz, Mohammad Sadil Khan, Anis Kacem, Gleb Gusev, and Djamila Aouada. Sharp challenge 2023: Solving cad history and parameters recovery from point clouds and 3d scans. overview, datasets, metrics, and baselines. In *CVPRW*, 2023. 6, 9, 11

[36] OpenAI. Gpt-4 technical report, 2024. 2, 3, 4, 6, 7, 8, 16

[37] Thomas Paviot. Pythonocc - 3d cad/bim/plm/cam framework, 2022. 9

[38] Zeju Qiu, Weiyang Liu, Haiwen Feng, Zhen Liu, Tim Z Xiao, Katherine M Collins, Joshua B Tenenbaum, Adrian Weller, Michael J Black, and Bernhard Schölkopf. Can large language models understand symbolic graphics programs? *arXiv preprint arXiv:2408.08313*, 2024. 2, 7, 16

[39] Daxuan Ren, Jianmin Zheng, Jianfei Cai, Jiatong Li, and Junzhe Zhang. Extrudenet: Unsupervised inverse sketch-and-extrude for shape parsing. In *Eur. Conf. Comput. Vis.*, pages 482–498. Springer, 2022. 2, 3

[40] V Sanh. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019. 3

[41] Ari Seff, Wenda Zhou, Nick Richardson, and Ryan P Adams. Vitruvion: A generative model of parametric cad sketches. In *ICLR*, 2022. 3

[42] Gopal Sharma, Difan Liu, Subhransu Maji, Evangelos Kalogerakis, Siddhartha Chaudhuri, and Radomír Měch. Parsenet: A parametric surface fitting network for 3d point clouds. In *Eur. Conf. Comput. Vis.*, pages 261–276. Springer, 2020. 3

[43] Tarek M Sobh, J Owen, C Jaynes, Mohamed Dekhil, and Thomas C Henderson. Industrial inspection and reverse engineering. In *Proceedings of 1994 IEEE 2nd CAD-Based Vision Workshop*, pages 228–235. IEEE, 1994. 1

[44] Qwen Team. Qwen2-1.5b. `https://huggingface.co/Qwen/Qwen2-1.5B`, 2024. Accessed: Nov. 2024. 9

[45] Mikaela Angelina Uy, Yen-Yu Chang, Minhyuk Sung, Purvi Goel, Joseph G Lambourne, Tolga Birdal, and Leonidas J Guibas. Point2cyl: Reverse engineering 3d objects from point clouds to extrusion cylinders. In *IEEE Conf. Comput. Vis. Pattern Recog.*, pages 11850–11860, 2022. 1, 2, 3, 6

[46] Xiaogang Wang, Yuelang Xu, Kai Xu, Andrea Tagliasacchi, Bin Zhou, Ali Mahdavi-Amiri, and Hao Zhang. Pie-net: Parametric inference of point cloud edges. *Adv. Neural Inform. Process. Syst.*, 33:20167–20178, 2020. 3

[47] Karl DD 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 construction from human design sequences. *ACM TOG*, 40(4):1–24, 2021. 2, 3, 4, 6, 11

[48] Rundi Wu, Chang Xiao, and Changxi Zheng. Deepcad: A deep generative network for computer-aided design models. In *IEEE Conf. Comput. Vis. Pattern Recog.*, pages 6772–6782, 2021. 2, 3, 4, 6, 7, 9, 10, 11, 12, 16

[49] Runsen Xu, Xiaolong Wang, Tai Wang, Yilun Chen, Jiangmiao Pang, and Dahua Lin. Pointllm: Empowering large language models to understand point clouds. In *ECCV*, 2024. 2, 8

[50] Xianghao Xu, Wenzhe Peng, Chin-Yi Cheng, Karl DD Willis, and Daniel Ritchie. Inferring cad modeling sequences using zone graphs. In *IEEE Conf. Comput. Vis. Pattern Recog.*, pages 6062–6070, 2021. 2, 3

[51] Xiang Xu, Karl DD Willis, Joseph G Lambourne, Chin-Yi Cheng, Pradeep Kumar Jayaraman, and Yasutaka Furukawa. Skexgen: Autoregressive generation of cad construction sequences with disentangled codebooks. In *ICML*, pages 24698–24724. PMLR, 2022. 3, 9

[52] Xiang Xu, Pradeep Kumar Jayaraman, Joseph G Lambourne, Karl DD Willis, and Yasutaka Furukawa. Hierarchical neural coding for controllable cad model generation. *ICML*, 2023. 3, 6, 12

[53] Le Xue, Mingfei Gao, Chen Xing, Roberto Martín-Martín, Jiajun Wu, Caiming Xiong, Ran Xu, Juan Carlos Niebles, and Silvio Savarese. Ulip: Learning a unified representation of language, images, and point clouds for 3d understanding. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 1179–1189, 2023. 2

[54] Le Xue, Ning Yu, Shu Zhang, Artemis Panagopoulou, Junnan Li, Roberto Martín-Martín, Jiajun Wu, Caiming Xiong, Ran Xu, Juan Carlos Niebles, and Silvio Savarese. Ulip-2: Towards scalable multimodal pre-training for 3d understanding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 27091–27101, 2024. 2

[55] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. Qwen2 technical report, 2024. 2, 4, 5

[56] Fukun Yin, Xin Chen, Chi Zhang, Biao Jiang, Zibo Zhao, Jiayuan Fan, Gang Yu, Taihao Li, and Tao Chen. Shapegpt: 3d shape generation with a unified multi-modal language model, 2023. 2

[57] Yang You, Mikaela Angelina Uy, Jiaqi Han, Rahul Thomas, Haotong Zhang, Suya You, and Leonidas Guibas. Img2cad: Reverse engineering 3d cad models from images through vlm-assisted conditional factorization. *arXiv preprint arXiv:2408.01437*, 2024. 3

[58] Fenggen Yu, Qimin Chen, Maham Tanveer, Ali Mahdavi Amiri, and Hao Zhang. D2csg: Unsupervised learning of compact csg trees with dual complements and dropouts. *Advances in Neural Information Processing Systems*, 36: 22807–22819, 2023. 3

[59] Fenggen Yu, Qimin Chen, Maham Tanveer, Ali Mahdavi Amiri, and Hao Zhang. D2csg: Unsupervised learning of compact csg trees with dual complements and dropouts. *Advances in Neural Information Processing Systems*, 36, 2024. 3

[60] Haocheng Yuan, Jing Xu, Hao Pan, Adrien Bousseau, Niloy J Mitra, and Changjian Li. Cadtalk: An algorithm and benchmark for semantic commenting of cad programs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3753–3762, 2024. 2

[61] Renrui Zhang, Ziyu Guo, Wei Zhang, Kunchang Li, Xupeng Miao, Bin Cui, Yu Qiao, Peng Gao, and Hongsheng Li. Pointclip: Point cloud understanding by clip. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8552–8562, 2022. 2

[62] Zibo Zhao, Wen Liu, Xin Chen, Xianfang Zeng, Rui Wang, Pei Cheng, BIN FU, Tao Chen, Gang YU, and Shenghua Gao. Michelangelo: Conditional 3d shape generation based on shape-image-text aligned latent representation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. 2, 5

[63] Xiangyu Zhu, Dong Du, Weikai Chen, Zhiyou Zhao, Yinyu Nie, and Xiaoguang Han. Nerve: Neural volumetric edges for parametric curve extraction from point cloud. In *IEEE Conf. Comput. Vis. Pattern Recog.*, pages 13601–13610, 2023. 3

[64] Xiangyang Zhu, Renrui Zhang, Bowei He, Ziyu Guo, Ziyao Zeng, Zipeng Qin, Shanghang Zhang, and Peng Gao. Pointclip v2: Prompting clip and gpt for powerful 3d open-world learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 2639–2650, 2023. 2