EUROGRAPHICS 2026 / B. Masia and J. Thies
(Guest Editors)

Volume 45 (2026), Number 2

CADrawer : Autoregressive CAD Generation from 3D Sketches

SUBMISSION ID : 1070

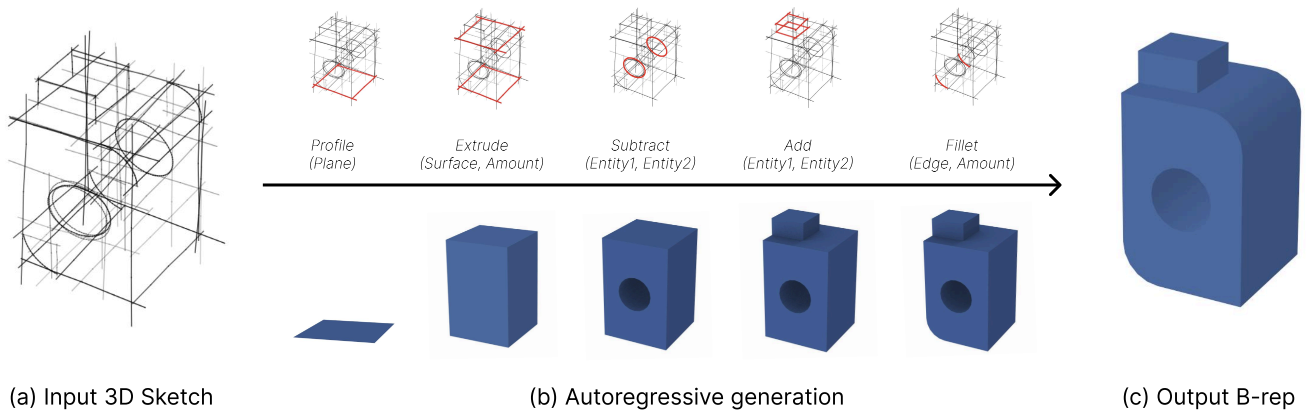


Figure 1: *Our system takes as input a 3D sketch, and autoregressively generates a CAD program that produces the intended shape.*

Abstract

In professional design workflows, designers often begin by creating sketch drawings before converting them into CAD programs. However, prior work on automatically interpreting these sketches has been limited to simplified inputs and fails to account for construction lines that are ubiquitous in real-world drawings. We present CADrawer, a system that translates 3D sketches into CAD programs using an autoregressive approach, leveraging construction lines as a rich source of information for recovering intermediate CAD operations. At each step, CADrawer predicts the next modeling operation and its parameters based on a graph-based representation of the sketch, which explicitly encodes spatial and temporal relationships between strokes. To improve generation quality, the system maintains multiple candidate programs in parallel, and a learned value function evaluates these partial programs to guide the search toward the most promising candidates. CADrawer is designed as a complement to 3D sketching interfaces, building on existing methods that creates 3D sketches. We evaluate our method across several datasets, including those containing dense construction lines and cases without ground-truth B-rep shapes.

CCS Concepts
• ***Computing methodologies*** $\rightarrow$ ***Shape modeling;***

1. Introduction

Computer-Aided Design (CAD) is a widely adopted standard for creating 3D shapes across various industries. CAD models are typically represented as programs consisting of a sequence of parametric modeling operations, such as extruding a 2D profile to create a solid block or rounding an edge to create a fillet. The parameters of these operations offer precise control on the dimensions of the geometry produced when executing the programs.

However, creating CAD models requires significant expertise in both planning the sequence of modeling operations and selecting them in feature-rich software interfaces. Meanwhile, sketching offers a quick and flexible way for designers to visualize the 3D shapes they have in mind, and to plan how to construct these shapes in CAD modeling. Prior research [LPBM20, HLMB22] and design educators [Hen12, Sto08] point to strong similarities between the steps designers follow when sketching 3D shapes, and the operations they use to model in CAD software. In this paper, we present a method that exploits these similarities to translate industrial design sketches into CAD programs.

Prior works [LPBM22, LPBM20, SLX*25] have introduced systems that recognize CAD operations from sketches, but these methods are restricted to sketches containing only feature lines, where each stroke directly corresponds to an edge of the final geometry

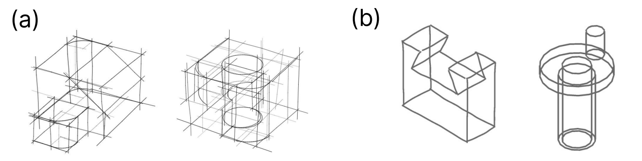


Figure 2: *Examples of sketches that our system can process (a), compared to the examples of sketches handled by previous work [LPBM22, LPBM20, SLX*25] (b)*

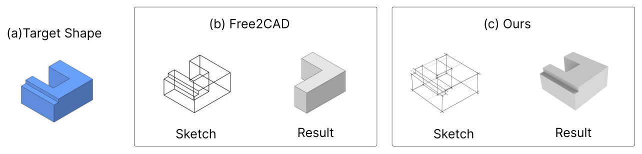


Figure 3: *We illustrate a case where Free2CAD [LPBM22] fails to reconstruct the target shape (a) when relying solely on the feature lines shown in (b). Since the strokes corresponding to the subtraction operation are absent, the method cannot recover the correct modeling process. In contrast, sketching the same shape with construction lines provide additional information about intermediate structures (c), which our method exploits to successfully reconstruct the intended shape.*

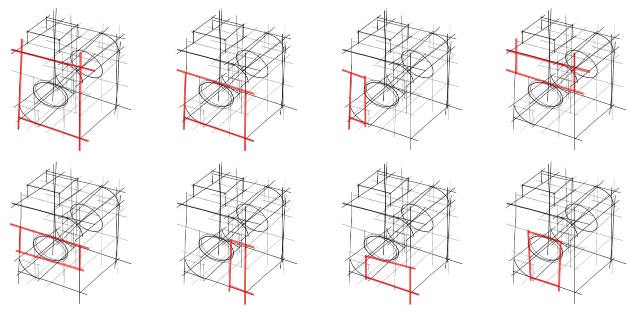


Figure 4: *Construction lines can form multiple loops on the same surface, increasing ambiguity and complexity in sketch interpretation.*

(Figure 2). Relying solely on feature lines makes it difficult to recover complex sequences of additive and subtractive operations, since multiple edits can occur within the same spatial region and the resulting lines may not appear in the finished shape. In contrast, real-world sketches often include construction lines — auxiliary strokes that designers use to outline primary object parts with simple primitives (e.g., cuboids, cylinders) before refining details [GSH*19]. Such lines frequently represent intermediate shapes in the design process or help establish perspective. While construction lines do not appear in the final geometry, many reveal crucial information about the process creating the final shape. Figure 3 gives a typical example of a shape that previous method [LPBM22] fails to recover using only feature lines.

The main challenge arises from the fact that construction lines often outnumber feature lines, leading to visual clutter without a direct correspondence to the final shape. As the number of strokes grows, the number of possible loops formed by construction lines also increases, as shown in Figure 4. In addition, designers may omit or repeat strokes, which further increases ambiguity in real-world sketches. The problem we address is to infer CAD operations and their parameters from such noisy and cluttered sketches. Our system operates on 3D sketches, as intersections and planar cycles are more easily detected in 3D than in 2D. These 3D sketches are becoming increasingly accessible through sketch-based modeling interfaces [SKSK09, WB25], sketch reconstruction methods [GHL*20, HGSB22], and VR interfaces [YDSG21]. We demonstrate our approach on 3D sketches that we have created using an existing drawing interface and accompanying reconstruction algorithm [WB25, HGSB22], even though our algorithm could apply to other sources of 3D sketches.

Our key observation is that the geometric relationships between the strokes convey rich information about the underlying 3D structure. In contrast to prior work that relies on Transformer-based models to discover stroke interactions [LPBM22], we explicitly encode geometric relationships in a graph where nodes represent sketch entities and edges encode spatial and temporal ordering between these entities. This custom representation allows us to adopt a lightweight graph neural network for analyzing the sketch and predicting the CAD operations. Furthermore, we augment the graph with information from the generated geometry, which provides both spatial and programmatic context.

We adopt an autoregressive approach that predicts a single operation and its corresponding parameters at each step. This sequential formulation allows the model to postpone uncertain decisions and use the progressively built shape to guide more informed and immediate predictions. However, like previous methods, this approach is prone to error accumulation. We maintain a set of candidate programs in parallel and apply Sequential Monte Carlo (SMC) to resample the best candidates. We use a learned value function that evaluates each partial program and concentrates computational resources on the most promising ones during the SMC process.

In summary, our system is the first to tackle the challenge of interpreting sketches that include both feature and construction lines. It takes as input 3D sketches and autoregressively generates CAD programs that can be executed to produce shapes aligned with the input. We evaluate our system on both synthetic and hand-drawn sketches spanning a range of complexities. We will release our code upon acceptance.

Our main contributions are:

- An autoregressive framework for translating 3D sketches into CAD programs.
- A graph-based representation of 3D sketches that captures geometric relationships between sketch entities.
- A learned value function that evaluates CAD programs by estimating their potential to reproduce the depicted shape.

2. Related Works

Our work builds on two complementary streams of research — sketch-based modeling and CAD program synthesis. We refer to recent surveys for extensive discussions of these two domains [LB25, RGJ*23].

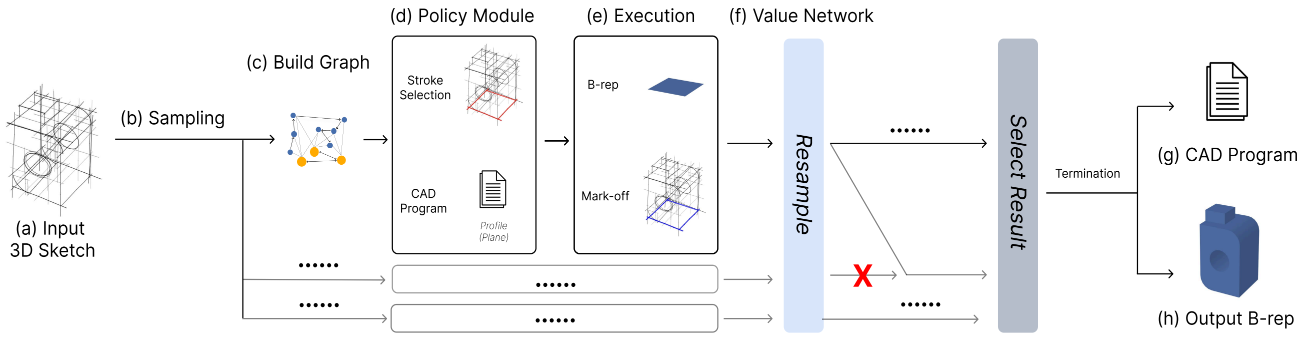


Figure 5: *Our system takes as input **(a)** a 3D sketch and performs autoregressive generation to produce **(g)** a CAD program, and **(h)** the resulting B-rep shape by executing the program. We create multiple samples that run in parallel, which are resampled after each step to maintain diversity and guide the generation progress. At each autoregressive step, we first build a graph **(c)** representing the current state of the reconstruction (Section 4), and then the policy module **(d)** predicts a CAD operation and identifies the strokes used to derive its parameters (Section 5). The current program is then executed and compared with the input 3D sketch to mark off the strokes that are already represented in the current program **(e)**. This feedback is used as input for the next step. After each step, the value function **(f)** estimates the likelihood of success for the current program state, allowing us to focus the search on more promising samples (Section 6).*

Sketch-based modeling. The field of sketch-based modeling has matured to offer a broad range of interactive and automatic approaches to create 3D shapes from 2D drawings. Optimization-based algorithms tackle this challenge by imposing geometric constraints between lines, such as parallelism and orthogonality [LS96], planarity [LCLT08, YLT13], symmetry [CSMS13, PCV16]. While early methods were limited to polyhedral shapes and clean drawings, later algorithms have been extended to curved objects [XCS*14, SKSK09], and sketches with oversketching and construction lines [GHL*20, HGSB22]. Building on this body of work, we assume that our input is a 3D sketch created with these methods. Taking 3D sketches as input facilitates the detection of sketch entities and their spatial relationships, allowing us to focus on recognizing CAD operations from such entities.

We contribute to the family of works that recognize parametric shapes from sketches [HKYM16, NGDGA*16, LPBM20, LPBM22, PMKB23, SLX*25]. In particular, our method is closest to Free2CAD [LPBM22] that autoregressively identifies groups of strokes that depict CAD operations and derive their parameters. However, both works are limited to simple, clean contour drawings that only contain feature lines that appear in the final shape. In contrast, the design sketches we target contain construction lines, which provide additional information about intermediate CAD operations, but also make the identification process more challenging.

Learning to Recover CAD Programs Our work also relates to the more general goal of reverse engineering CAD models from diverse input, such as voxel grids [SGL*18, TLS*19, LWJ*22], point clouds [WXW18, DIP*18, WXZ21, LOWS23, GLP*22, SLK*20, RDM*24], boundary representations [XPC*21] or others [CF25, WZW*24]. Working on sketches gives us a unique advantage, as the drawing sequence we take as input not only depicts the final shape envisioned by the designer, it also describes how the designer plans to construct it. This additional information helps recovering the ordering of CAD operations, as observed by prior work on sketch-based modeling [LPBM20, LPBM22].

Inspired by prior on deep learning for CAD, we propose to represent 3D sketches with a graph structure that encodes stroke ordering, stroke intersections, and stroke loops. This choice aligns with the inherent nature of CAD boundary representations (B-reps), where graphs naturally capture the relationships between faces, edges, and vertices. Many previous works have proposed their own graph representations tailored to the specific needs of their tasks [XPC*21, CRN*22, WJC*22, JHC*21, JNK*23]. Our representation jointly encodes the 3D sketch and the B-rep generated by executing the CAD program, which enables effective mapping between our input and output while providing spatial context for program generation.

Our approach also builds on ideas from previous works in program synthesis [ENP*19, ERSLT18, CLS19, TLS*19, KMP*18] that incorporate execution-based feedback into autoregressive generation. Specifically, we adopt an autoregressive approach to capture the sequential nature of CAD programs, where later operations often depend on geometry generated in earlier steps. We extend this paradigm by executing the partial program at each step, comparing the resulting B-rep with the input 3D sketch, and using spatial feedback to guide the next prediction. This execution-feedback loop enables the system to remain aware of construction progress and avoid redundant operations.

3. Approach

Our system takes as input a 3D sketch—a set of 3D polylines, each represented by 10 sampled points—and outputs a CAD program that generates the intended 3D shape. We adopt an autoregressive generation process that adds one CAD operation token and its corresponding parameters at each step. Each autoregressive step consists of three actions. First, the system constructs a graph representing the current generation state (Section 4). Next, the policy module predicts the next CAD operation and selects the relevant subset of strokes or loops to determine its parameters (Section 5). Then, the system executes the current program to produce an updated B-rep and compares it against the input 3D sketch to identify which strokes have been explained.

We maintain multiple program samples in parallel. After each step, once all samples have finished execution, a learned value function evaluates their current states, and a resampling step reallocates computational resources to the most promising samples (Section 6).

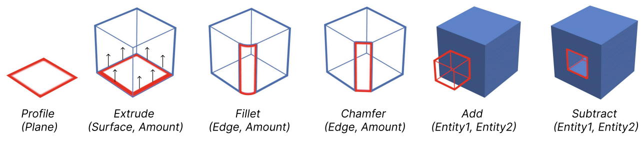


Figure 6: *Our system supports six operations: `profile`, `extrude`, `fillet`, `chamfer`, `add`, and `subtract`.*

Similar to previous sketch-to-CAD works [LPBM20, LPBM22, SLX*25] and many other CAD research efforts, our system supports four fundamental CAD operations: `profile`, `extrude`, `fillet`, and `chamfer` (Figure 6). Boolean operations emerge from the extrude direction. Extruding outward `add` material, while extruding inward `subtract` material.

4. Graph Representation

At each autoregressive step t, we construct a heterogeneous graph $G_t = (V, E)$ that encodes the spatial relationships between strokes, their sequential order, and the current state of the CAD program (Figure 7a,b). To capture the program state, we execute the partially generated CAD program to produce a B-rep and compare it against the input 3D sketch to identify which strokes have already been explained (Figure 7c). This comparison provides spatial grounding, as it is difficult for neural networks to perform spatial reasoning solely from symbolic program tokens. The resulting unified graph is in one-to-one correspondence with the evolving CAD program, ensuring that each program state has a unique graph representation. This graph serves as input to both the policy module and the value network, providing information from both the sketch and the CAD program.

Prior work such as [YZF*21] represents sketches as graphs where nodes correspond to sampled points and edges to stroke segments. However, this approach captures only local geometry and struggles with more complex sketch structures. In contrast, Free2CAD [LPBM22] models sketches as sequences using a Transformer-based architecture to capture temporal order of strokes, but neglects spatial relationships and incurs substantial computational costs (9 days of training reported). Concurrently to our work, Sketch2Seq [SLX*25] is based on a graph structure that encodes strokes as nodes and local and distant spatial relationships as edges, but it ignores stroke ordering and larger entities such as loops formed by successive strokes.

Our method combines the strengths of these approaches: we encode both sequential and spatial relationships in a unified graph structure using heterogeneous edge types. This allows for efficient processing with a lightweight graph neural network that can be trained within a few hours. Furthermore, our graph includes two types of nodes: *stroke nodes* and *loop nodes*. Loop nodes represent coplanar, closed groups of strokes that typically define profile regions for planar operations. These nodes ensure that the profile detection module can consistently identify closed, complete sketch planes. Another challenge is the *ambiguity of stroke roles*, where the purpose of a stroke may only become clear after earlier parts of the sketch are interpreted. Our graph representation addresses this by allowing each stroke to reason about its spatial and temporal neighbors and the usage status of those neighbors.

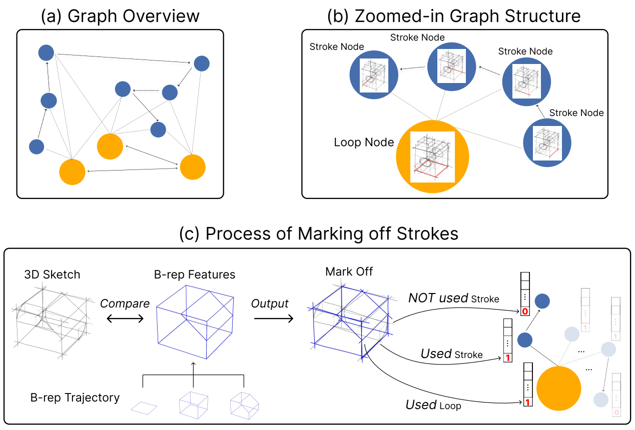


Figure 7: *An overview of the graph (a), and a zoomed-in view (b). Panel (c) shows the process of marking off strokes. We execute the CAD program incrementally to produce all intermediate shapes generated throughout the process. This is because certain edge features, especially those involved in `subtracts`, may not appear in the final shape. The resulting mark-off (in blue) indicates which strokes have been explained by the current program.*

4.1. Graph Nodes

The input 3D sketch is represented as a set of polylines, with each polyline sampled at 10 points. For each stroke, we fit a parametric function based on its geometry, including: straight lines, circular arcs, full circles, ellipses, and free-form curves. Each stroke node in our graph encodes the corresponding parametric function, the stroke's opacity, its type, and a binary label indicating whether it is used in the final B-rep. In contrast, each loop node contains only a binary indicator for B-rep usage. We provide additional details on the node feature representations in Appendix A.

4.2. Graph Edges

The graph edges capture both the spatial relationships between nodes and the temporal order of stroke execution. Stroke-order edges are defined directly from the sequence in which strokes are drawn, while all other edges are derived purely from geometric relations. To assess the contribution of each edge type, we perform an ablation study in Section 8.5. The edge categories are as follows:

- **Stroke-to-Stroke Edges:** Capture intersection between strokes in the 3D sketch.

- **Loop-to-Loop Edges:** Capture intersection, containment (which loop contains which), and perpendicular relationships between loops.
- **Stroke-to-Loop Edges:** Indicate which strokes constitute a particular loop.
- **Stroke-order Edges:** Capture the order in which strokes were drawn.

5. Policy Module

Our system autoregressively generates a CAD program $\mathcal{P} = \{p_t\}_{t=1}^{T}$, where each $p_t = (o_t, \theta_t)$ denotes a CAD operation o_t and its associated parameters θ_t. At each timestep t, the policy module takes as input the graph constructed in Section 4 and performs three tasks: (1) predicting the next CAD operation o_t (Section 5.2); (2) selecting the relevant strokes from $\mathcal{S}$ (Section 5.2); and (3) inferring the operation parameters θ_t based on the selected strokes (Section 5.3).

5.1. Graph Encoder

We use a shared Graph Convolutional Network (GCN) encoder to compute node embeddings from the input graph G_t. These embeddings are then fed into task-specific decoders for different tasks. We provide detailed architecture of the network in Appendix B.

5.2. Task-Specific Decoders

We design different decoders tailored to different tasks, each following a specific pipeline (see Figure 8), and train them separately.

5.2.1. (a) Operation prediction.

To predict the next CAD operation token, we perform cross-attention between the program embedding (as query) and the graph embeddings (as key and value), thereby annotating the program with geometric context. We then apply self-attention over the annotated program embedding (the `[CLS]` token) to aggregate information and produce the next program token:

$$\mathcal{L}_{\text{op}} = -\sum_{i=1}^{|\mathcal{O}|} y_i \log \hat{y}_{\text{op},i}. \tag{1}$$

Our loss function is the standard cross-entropy loss, which penalizes the model when it assigns low probability to the correct operation token.

5.2.2. (b–d) Stroke (or Loop) Feature Selection.

For operations that require geometric input, such as selecting a loop for `Profile`, strokes for `Extrusion`, or strokes for `Fillet/Chamfer`, we perform binary classification over the relevant nodes. For each node v, we compute a selection probability by minimizing the following focal loss:

$$\mathcal{L}_{\text{stroke}} = -\sum_{i=1}^{|\mathcal{S}|} \alpha_i (1-\hat{y}_{\text{stroke},i})^{\gamma} y_i \log \hat{y}_{\text{stroke},i}, \tag{2}$$

where $y_i \in \{0,1\}$ indicates whether node i is selected, $\alpha_i = 1.0$, and $\gamma = 1.5$. The focal loss [LGG*17] mitigates class imbalance by down-weighting easy negatives, which is important for our case since only a small fraction of nodes are typically selected at each step.

The loss function in Eq. (2) serves as the common objective for all geometric selection tasks. The specific pipeline for constructing the candidate set of graph nodes, however, differs by task, as described below:

- **Profile selection (b):** An MLP is applied to the loop embeddings, followed by binary classification. The loop with the highest probability is selected.
- **Extrusion (c):** During graph construction, sketch strokes corresponding to previously used sketch operations are masked, so the graph encodes which strokes are already chosen. The encoder produces graph embeddings, and an MLP predicts which strokes are used for extrusion. A new graph is then built with these strokes masked, re-encoded, and the decoder selects the face created by the extrusion.
- **Fillet and chamfer selection (d):** An MLP is applied directly to the stroke embeddings, followed by binary classification. The contributing strokes are then selected.

5.2.3. (e) Value network.

After generating graph embeddings, we compute cross-attention between the graph embeddings and their mean-pooled representation. This enables the network to capture both global and local features of the graph. The resulting representation is passed through an MLP to regress to a single scalar value.

5.3. Finding Operation Parameters

Given the strokes (or loops) selected for each operation, we extract continuous values required to execute that operation. A major challenge is that the input strokes are sketches that are inherently imprecise, making it difficult to recover exact parameter values directly. To address this, we employ a set of geometric algorithms to infer the parameters, as detailed in Appendix C.

6. SMC Based Program Sampling

Performing the entire autoregressive generation process in a single shot is challenging. First, errors can accumulate across steps, compounding over time. Second, 3D sketches are often ambiguous so that multiple valid interpretations may exist, and later decisions may depend on earlier ones. To capture this uncertainty and maintain a diverse set of plausible solutions, we adopt a Sequential Monte Carlo (SMC) framework that maintains a set of samples CAD programs, referred to as *particles*.

All particles are initialized from the same state: the empty program. At each timestep t, each particle samples its next step from the policy module, which involves predicting the next operation token and selecting the corresponding strokes. This procedure defines the prior distribution:

$$p\left(x_{0:t}^{(i)}\right) = \prod_{k=1}^{t} p\left(x_k^{(i)} \mid x_{0:k-1}^{(i)}\right),$$

where $x_k^{(i)}$ denotes the program step chosen at time k by particle i.

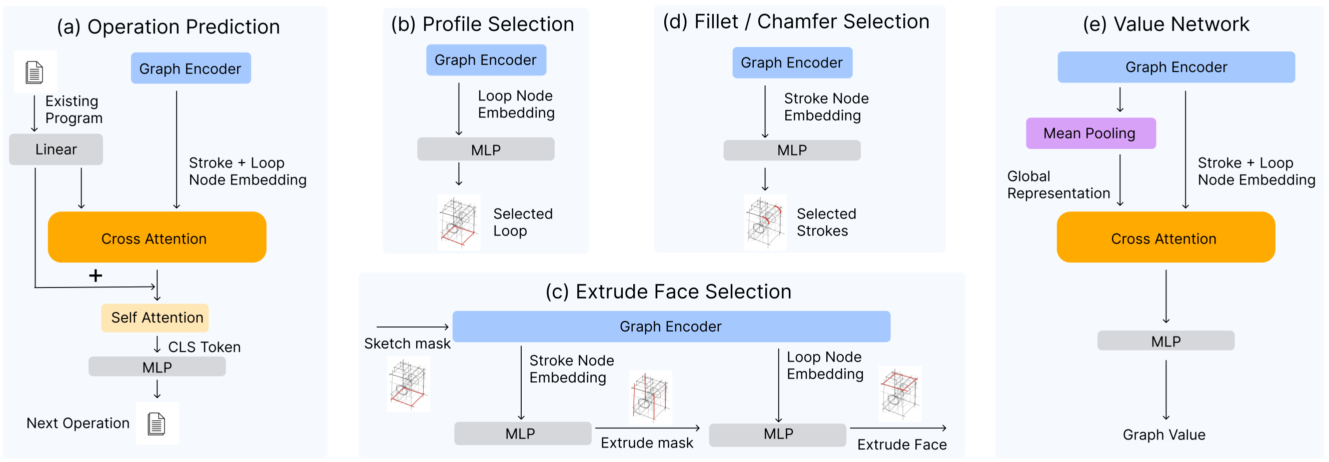


Figure 8: *Overview of the decoder architecture. Each submodule is responsible for a specific task: (a) operation prediction, (b) profile selection, (c) extrude face selection, (d) fillet/chamfer selection, and (e) value network.*

SMC then approximates the posterior distribution $p(x_{0:t} \mid y)$, where y is input graph G_t. As directly computing this posterior is intractable, SMC resamples the particles based on a learned value function $V(x_{0:t})$ (Section 6.1). This resampling helps recover from early mistakes and maintain diversity among plausible particles , which is particularly important for complicated sketches. In Figure 9 we show an example of this process.

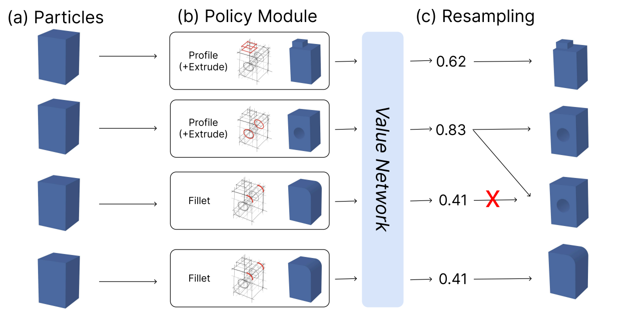


Figure 9: *We present an example of resampling using SMC. After all particles pass through the policy module, the value network assigns each of them a score. The SMC then resamples based on these scores, shifting the distribution toward particles with higher likelihood.*

6.1. Value Function

We need a scoring function that evaluates how well a candidate CAD program matches the target sketch. Since different execution orders of CAD operations can produce the same final B-rep, this value function must be order-invariant.

Previous works on CAD generation often evaluate their results by computing the Chamfer Distance between the generated B-rep and inputs such as voxels [UyCS*22, KSA23], point clouds [GLP*22, ZHFL23, LCP*24], or meshes [GXL13]. In contrast, directly comparing our generated B-rep with the input 3D sketch is not meaningful. Such a comparison only reveals which strokes have been explained by the program. But many construction lines are not intended to appear in the final shape, and the input sketch itself is sparse.

Instead, we evaluate the generation process by computing the Chamfer distance between the generated B-rep and the ground-truth B-rep. However, during inference, the ground-truth shape is not available, making direct computation infeasible. To address this, we train a neural network that takes the current graph G_t as input and learns to predict a proxy for the Chamfer distance. This learned value function enables geometry-aware scoring of partial CAD programs without the ground truth during generation.

6.1.1. Immediate Value Estimation

A straightforward approach is to train our neural network to predict the Chamfer distance S_f of the current B-rep. However, as Chamfer distance is correlated with the volume of the shapes, operations that create larger volumes (e.g., `extrude`, which produces a solid block) might have greater impacts on the Chamfer distance than operations that modify smaller features (e.g., `fillet`, which rounds edges). In our experiments, we observe that the SMC sampling process with this immediate value estimation tend to favor samples that prioritized `extrude` operations, leading to a greedy search behavior.

6.1.2. Expected Value Estimation

A more principled way to evaluate a partial CAD program is by estimating the quality of its expected final output. Inspired by prior works such as AlphaGo [SHM*16, SSS*17], we construct a search tree that explores possible future executions from the current program state (Figure 10). The value of a partial program is then computed by aggregating the values of all possible completions, weighted by their probabilities. We define the value of a state s as:

$$V(s) = \sum_{a \in A} P(a|s)V(s')$$

where:

- $V(s)$ is the value of the current state.

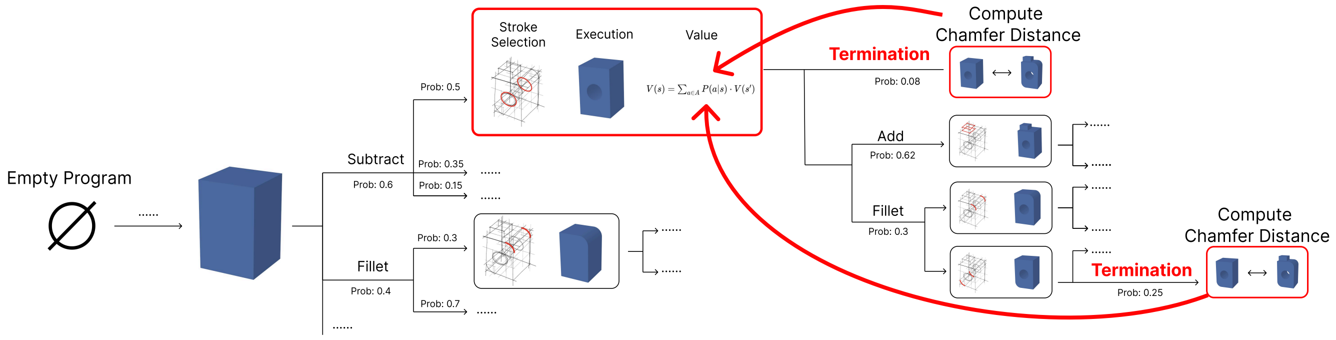


Figure 10: *We build a tree from a a partial CAD program by simulating future actions. Each branch represents a possible choice by the policy module. Non-terminal states' values are based on their child nodes weighted by probabilities. Terminated states are evaluated using Chamfer distance to the ground truth shape.*

- A is the set of possible operations from s.
- $P(a|s)$ is the probability of writing program a from state s.
- s' is the next state obtained by applying a to s.
- $V(s')$ is the value of the next state, or the Chamfer distance if it is an termination state.

However, constructing a complete tree that explores all possible executions of the system is computationally infeasible. We approximate this process using Monte Carlo Tree Search (MCTS), which focuses exploration on high-impact branches. Implementation details of our MCTS algorithm are provided in Appendix D.

To train the value network, we use the search trees generated by our MCTS procedure to construct a dataset that provides estimated values for program states at various stages of execution. We adopt the same graph encoder (detailed in Appendix B) to produce node embeddings, then passed through a value decoder (Figure 8) to predict a scalar value representing the estimated quality of the current state. We train the value network using a contrastive loss that encourages higher scores for better programs:

$$\mathcal{L}_{\texttt{value}} = \max(0, m - y \cdot (S_1 - S_2)),$$

where S_1 and S_2 are the predicted scores, $y \in \{1, -1\}$ indicates which program is better, and $m = 0.2$ is the margin.

7. Implementation

7.1. Dataset

We develop a novel method to generate noisy synthetic 3D sketches that imitate human sketches (detailed in Appendix E) and prepare two datasets using it. The first dataset (Figure 12 and Figure 14), introduced by [HLMB22], consists of 1361 CAD program and 3D sketch pairs and includes `profile`, `extrude`, and `fillet` operations. Each program contains exactly 8 operations, and the resulting sketches have an average of 78.6 strokes, with a minimum of 35 and a maximum of 143 strokes. To increase diversity and complexity, we procedurally generate a second dataset comprising 4000 CAD program and 3D sketch pairs (Figure 13 and Figure 15), covering all four basic operations: `profile`, `extrude`, `fillet`, and `chamfer`. Program lengths range from 3 to 15 operations, with an average of 9.2. The resulting sketches vary from 17 strokes at the simplest end to 307 strokes at the most complex, with an average of 122.3 strokes.

Both datasets are divided into 80% for training and 20% for validation. They feature diverse designs (exampled in Figure 14, 15) and differs in program length, program patterns, spatial relationships between strokes, as well as in how feature lines and construction lines are drawn. We train on these datasets jointly to highlight generality. In Section 8, we present results from training both separately and jointly (by randomly merging them into a single combined dataset), demonstrating our system's ability to generalize across a wide range of sketching styles.

7.2. Network Training and Inference

We implement our neural networks in PyTorch Geometric and will release the code upon acceptance. Training is performed on an NVIDIA GeForce RTX 4090 GPU: policy networks train in $\sim$2 hours, and the value network in $\sim$10 hours. At inference time, our system generates a CAD program (9 operations) in $\sim$30 seconds using 30 parallel particles in the SMC framework.

8. Results and Evaluation

8.1. Baseline Method : Order Based Reconstruction

We implemented a baseline algorithm that processes strokes in the order they were drawn. In this approach, strokes are sequentially added, and whenever they form a closed loop, the algorithm groups them into a sketch loop. When such loops correspond to higher-level entities (e.g., a cuboid), the algorithm generates the corresponding sketch and extrude operations to construct the intended geometry.

However, this approach faces two major challenges. First, artists often draw in inconsistent order, sometimes revisiting earlier parts of the sketch. Second, sketches frequently include construction lines, which can form loops with feature lines. This algorithm often mistakenly interpret these as valid sketches, resulting in errors. To evaluate this method, we selected 100 short programs from Dataset B (each containing only six operations). The baseline succeeded in generating only 1 out of 100 shapes, clearly illustrating its limitations.

8.2. Baseline Method : Stroke Filtering as Preprocessing

Another baseline method we consider is a two-stage pipeline. The first stage selects strokes that either appear in the final shape or in intermediate shapes, since these strokes can help generate the entire CAD generation process. Such strokes include both feature lines and a subset of construction lines. Our objective is to use only these selected strokes to predict the CAD program, thereby reducing the burden on the network. Specifically, we train a network to perform binary classification of strokes, separating those that are ever used in the shape's generation history (i.e present in intermediate shapes or the final shape) from those that are not (i.e. construction lines used solely for perspective correction). This classifier adopts the same graph encoder as our main pipeline to compute node embeddings, followed by a multilayer perceptron (MLP) that operates on the stroke nodes.

We evaluated this approach on 500 shapes sampled from Dataset B. The preprocessing network achieved an accuracy of 86.2% in distinguishing between the two categories of lines. However, only 173/500 examples retained all the lines required to fully generate the program. For the remaining 327/500 examples, recovering the correct program was difficult regardless of the generation algorithm. This stroke pre-processing does not work well because it is inherently challenging to determine which construction lines are essential for the generation process in a single-shot prediction. In contrast, our method (proposed in this work) addresses this challenge through an autoregressive formulation, where later predictions can build on earlier ones, making it easier to capture the necessary lines for program recovery. We provide example results of predicting lines that are used in the shape's generation history in Figure 11.

8.3. Overall Performance

We train our network on the two datasets both separately and jointly. Joint training on Dataset A and Dataset B enables broader generalization, but it also introduces challenges due to stylistic inconsistencies between the datasets. For example, Dataset B often uses diagonal lines to denote profile planes, whereas Dataset A does not (Figure 14, Figure 15). Such differences can confuse the network, since identical operations are represented with different visual cues. Nevertheless, our system remains capable of making valid predictions by reasoning about underlying spatial relationships rather than relying solely on dataset-specific patterns. This indicates that the model learns to infer higher-level geometric intent, contributing to its robustness.

To assess shape quality, we compute the Chamfer distance between the generated shape and the ground-truth shape in the validation set, using 300 uniformly sampled surface points. A generation is considered successful if the Chamfer distance is less than 1% of the bounding box diagonal of the ground truth shape. We also present several failure cases and their underlying causes in Figure 16.

We observe that the value function often struggles to distinguish fine-grained shape differences, particularly those involving small features such as `fillet` or `chamfer` operations. To mitigate this limitation, our system returns the top-3 predicted shapes, ranked by the value function, and allows users to select their preferred result.

Table 1: *Top-3 results success rate (%) with different sampling methods.*

Value Function	**Dataset A**	**Dataset B**	**Joint (A + B)**
No Sampling / No Value Function	59.0%	67.0%	48.0%
Immediate Value Function	82.0%	89.0%	80.0%
MCTS based Value Function	**82.0%**	**91.0%**	**82.5%**

Table 2: *Accuracy (%) for operation prediction and corresponding strokes (or loops) selection across different dataset setups.*

Task Type	**Dataset A**	**Dataset B**	**Joint (A + B)**
`Profile`	88.7%	94.2%	**82.2%**
`Extrude`	94.2%	97.4%	**93.3%**
`Fillet`	89.6%	99.6%	**94.5%**
`Chamfer`	/	82.7%	/
`Next Operation`	99.7%	89.9%	**92.1%**

In Table 1, we compare the effectiveness of three sampling strategies: (1) a baseline without resampling, (2) SMC sampling with resampling based on an immediate value function (Section 6.1.1), and (3) SMC sampling guided by a value function trained to estimate the expected final outcome (Section 6.1.2).

The value function trained on expected final values does not provide any improvement over the immediate value function on Dataset A, whereas it shows a more noticeable benefit on Dataset B. This is likely because all programs in Dataset A follow a fixed operation sequence. As a result, greedy strategies that prioritize high impact operations like extrusion do not lead to incorrect programs.

8.4. Accuracy on Individual Tasks

We further assess the accuracy of individual modules (Table 2), covering operation prediction and stroke selection for `profile`, `extrude`, `fillet`, and `chamfer`. Chamfer accuracy is not reported for Dataset A, since it contains no chamfer operations, and is also omitted for joint training, as the results are identical to those of Dataset B. A prediction is considered correct only if all corresponding strokes (or loops) are selected.

8.5. Ablation Study: Graph Design

We examine our graph design by removing different graph edge types and record the network's performance on `profile`, `extrude` and `fillet` stroke selections on Dataset A. We show in Table 3, that removing any of these graph edges would lead to a decrease in certain tasks. Additionally, we experiment with a graph

Table 3: *Ablation study of graph design. We report average accuracy for* `Profile`*,* `extrude`*, and* `fillet` *selection tasks.*

Edge Type Removed	`Profile`	`Extrude`	`Fillet`
Full Graph	**88.7**	**94.2**	**89.6**
Stroke-intersect-Stroke	86.7%	65.1%	82.1%
Loop-perpendicular-Loop	81.1%	84.4%	87.7%
Loop-contains-Loop	69.3%	93.6%	89.2%
Stroke-to-Loop	82.3%	13.5%	85.6%
Stroke Order	67.2%	92.8%	70.4%
No Loop Nodes	45.6%	/	/

Table 4: *Top-3 results success rate (%) on Dataset B across different program lengths and numbers of SMC particles.*

Particles	**< 5 Step**	**5–7 Step**	**8–10 Step**	**11–15 Step**
30 Particles	99.2%	94.4%	82.9%	38.5%
50 Particles	99.2%	95.1%	83.9%	48.0%
100 Particles	99.2%	95.1%	87.4%	52.1%

that contains only stroke nodes. In this setting, the `profile` prediction is reformulated as identifying all strokes that form the profile region. The result of this variant is shown in the last row of the table.

8.6. Impact of Program Length and Sampling Budget

Our system's performance declines as the length of the target CAD program increases. Also larger number of particles during the SMC sampling process may improve results. We quantify this relationship using Dataset B (which has varying program length) in table 4.

Our system performance degrades significantly for programs longer than 10 steps, and especially beyond 12. These failure cases often involve missing smaller geometric features, such as `fillets` or `chamfers` (Figure 16). This degradation is likely due to several factors. First, longer programs correspond to sketches with more strokes, which inherently increases difficulty of the generation process. Second, autoregressive models are more prone to errors as sequence length increases. Third, the value estimation function performs less reliably on complicated densely sketches, which makes it hard to identify the 3 most promising final outputs.

8.7. Results Comparison with Free2CAD

We demonstrate that incorporating construction lines enables our method to reconstruct shapes that previous approaches, such as Free2CAD [LPBM22], fail to capture. The limitation arises because relying solely on feature lines makes it difficult to recover the complex sequence of additive and subtractive operations. Multiple edits may occur in the same spatial region and their traces are often absent in the final geometry.

In Figure 17, we highlight six examples taken from the Free2CAD supplemental material where the system was unable to generate the correct shapes. Since the original sketches contain only feature lines, important details are lost and the resulting reconstructions deviate from the intended design. To address this, we redrew the sketches with construction lines and applied our method. The inclusion of construction lines provides additional cues about intermediate structures in the modeling process, allowing our approach to accurately interpret them and produce final shapes that more closely match the sketch's intent.

8.8. Evaluating on Synthetic 2D Sketches

We qualitatively evaluate our method on synthetic 2D sketch drawings that are lifted to 3D to simulate noisy 3D sketches. Specifically, we first sample a subset of examples from CAD2Sketch [HLMB22], which generates 2D sketches from 3D shapes. We then uplift these sketches into 3D space using a symmetry-based algorithm [HGSB22]. As shown in Figure 18, our method successfully reconstructs the intended shapes.

8.9. User Study: Creating 3D Shape from 2D Sketches

We further evaluated our method on real-world 2D drawings. Specifically, we invited three students with limited prior CAD design experience and one student designer proficient in CAD design to create 2D sketches using an existing drawing interface equipped with a 3D lifting algorithm [WB25, HGSB22] (Appendix F). The resulting 3D sketches were then processed with CADrawer to generate 3D B-rep shapes. Each participant first received a brief 15-minute tutorial on the UI system (Appendix F) and on perspective drawing. They were then asked to produce three sketches of their choice in 2D space, which the system automatically uplifted into 3D sketches. While the participants exhibited diverse sketching habits, most of them used construction lines, consistent with our assumption (further discussed in Appendix G).

On average, participants spent about 21 minutes completing all three sketches. Students with limited prior CAD design experience found perspective drawing increasingly difficult as the sketches grew more complex, whereas the proficient student designer found our UI more intuitive and convenient. We then applied CADrawer to translate these 3D sketches into CAD programs, with the results presented in Figure 19.

We used a 5-point scale (1 = very unsatisfied/very different, 5 = very satisfied/highly similar) to evaluate participant feedback. Overall, participants reported a high level of satisfaction with both the sketching process and the automatic 3D lifting. The average similarity score was 4.6/5, indicating that the generated shapes were generally considered close to the original sketches. The system also received an average ease-of-use rating of 4.2/5. All participants with limited CAD experience agreed that it made creating 3D shapes easier than working directly with CAD software. In contrast, the proficient student designer found direct modeling in CAD software easier and more intuitive.

8.10. User Study: Expert Manual Shape Reconstruction

We dalso conducted a second user study to directly compare human experts in reconstructing 3D shapes from sketch drawing with the automated generation process of CADrawer. We invited three student designers from a prestigious design school, each proficient in CAD software and experienced in manual modeling workflows. In this study, participants were provided with six 3D sketches and asked to reconstruct the corresponding B-rep shapes manually, without the assistance of our system.

During the process, we observed that participants often struggled with sketches that involved complex modeling steps, particularly those requiring multiple subtraction operations. Overlapping strokes frequently created visual ambiguities, making it difficult to determine the intended sequence of operations. Participants also encountered challenges in accurately interpreting perspective from the sketches, whereas CADrawer automatically extracts precise geometric parameters from strokes.

At the same time, human designers demonstrated strong contextual reasoning and an ability to infer design intent beyond what was explicitly drawn. This often allowed them to avoid certain mistakes made by our system, such as misinterpreting partially drawn or ambiguous strokes. Notably, they could still infer correct parameters even when stroke values extended beyond the thresholds used by our algorithm.

We present a side-by-side comparison of the manually created shapes and the results generated by our system in Figure 20. This comparison illustrates the complementary strengths of expert human reasoning and automated CAD generation.

9. Conclusion

We introduce CADrawer, a new framework for generating CAD programs from 3D sketches.To the best of our knowledge, we are the first to leverage sketch construction lines as additional information for recovering intermediate CAD operations, which allows us to successfully manage complex sketches that challenged previous approaches. But construction lines bring additional clutter and ambiguity, which we handle with a combination of autoregressive prediction and Sequential Monte Carlo exploration.

A key challenge that remains is the lack of large-scale datasets of real-world sketch drawings and their corresponding B-rep programs. Acquiring such data is difficult due to the manual effort required. This limitation constrains the diversity of training samples and hinders the model's ability to generalize across varying sketching styles and modeling workflows.

One promising direction is to adopt a bootstrapped program synthesis strategy, as explored in [JWR22, EWN*21, JGMR23], where two networks are jointly trained to synthesize programs from sketches and generate human-like sketches from programs at the same time. This approach enables training an initial inference model on a small dataset, executing it to generate synthetic sketch-program pairs, and iteratively refining both the model and dataset via self-supervised learning.

References

[CF25] Chereddy S., Femiani J.: Sketchdnn: Joint continuous-discrete diffusion for CAD sketch generation. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)* (2025), PMLR, pp. 1–17. 17 pages, 63 figures. URL: https://arxiv.org/abs/2507.11579, doi:10.48550/arXiv.2507.11579. 3

[CLS19] Chen X., Liu C., Song D.: Execution-guided neural program synthesis. *ICLR* (2019). Presented at ICLR 2019. 3

[CRN*22] Colligan A. R., Robinson T. T., Nolan D. C., Hua Y., Cao W.: Hierarchical cadnet: Learning from b-reps for machining feature recognition. *Computer-Aided Design 147* (June 2022). doi:10.1016/j.cad.2022.103226. 3

[CSMS13] Cordier F., Seo H., Melkemi M., Sapidis N. S.: Inferring mirror symmetric 3d shapes from sketches. *Computer Aided Design 45*, 2 (2013). 3

[DIP*18] Du T., Inala J. P., Pu Y., Spielberg A., Schulz A., Rus D., Solar-Lezama A., Matusik W.: Inversecsg: Automatic conversion of 3d models to csg trees. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia 37*, 6 (2018). 3

[ENP*19] Ellis K., Nye M., Pu Y., Sosa F., Tenenbaum J., Solar-Lezama A.: Write, execute, assess: Program synthesis with a repl. *ICML* (June 2019). doi:10.48550/arXiv.1906.04604. 3

[ERSLT18] Ellis K., Ritchie D., Solar-Lezama A., Tenenbaum J. B.: Learning to infer graphics programs from hand-drawn images. *NeurIPS* (July 2018). doi:10.48550/arXiv.1707.09627. 3

[EWN*21] Ellis K., Wong C., Nye M., Sablé-Meyer M., Morales L., Hewitt L., Cary L., Solar-Lezama A., Tenenbaum J. B.: Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (2021), ACM, pp. 835–850. URL: https://doi.org/10.1145/3486607.3486750, doi:10.1145/3486607.3486750. 10

[GHL*20] Gryaditskaya Y., Hähnlein F., Liu C., Sheffer A., Bousseau A.: Lifting freehand concept sketches into 3d. *TOG 39*, 6 (Nov 2020). doi:10.1145/3414685.3417851. 2, 3

[GLP*22] Guo H., Liu S., Pan H., Liu Y., Tong X., Guo B.: Complexgen: Cad reconstruction by b-rep chain complex generation. *ACM Transactions on Graphics (SIGGRAPH) 41*, 4 (2022). 3, 6

[GSH*19] Gryaditskaya Y., Sypesteyn M., Hoftijzer J. W., Pont S., Durand F., Bousseau A.: Opensketch: A richly-annotated dataset of product design sketches. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)* (2019). 2

[GXL13] Gao S., Xu X., Lin C.: Topology reconstruction for b-rep modeling from 3d mesh in reverse engineering applications. *Computer-Aided Design 45*, 2 (2013), 496–507. URL: https://www.researchgate.net/publication/258712788_Topology_Reconstruction_for_B-Rep_Modeling_from_3D_Mesh_in_Reverse_Engineering_Applications, doi:10.1016/j.cad.2012.10.010. 6

[Hen12] Henry K.: *Drawing for product designers*. Laurence King Publishing, 2012. 1

[HGSB22] Hähnlein F., Gryaditskaya Y., Sheffer A., Bousseau A.: Symmetry-driven 3d reconstruction from concept sketches. *SIGGRAPH* (July 2022). doi:10.1145/3528233.3530723. 2, 3, 9, 15, 17

[HKYM16] Huang H., Kalogerakis E., Yumer E., Mech R.: Shape synthesis from sketches via procedural models and convolutional networks. *IEEE Transactions on Visualization and Computer Graphics (TVCG) 22*, 10 (2016), 1. 3

[HLMB22] Hähnlein F., Li C., Mitra N. J., Bousseau A.: Cad2sketch: Generating concept sketches from cad sequences. *ACM Transactions on Graphics (TOG) 41* (November 2022), 1–18. doi:10.1145/3550454.3555488. 1, 7, 9, 15

[JGMR23] Jones R. K., Guerrero P., Mitra N. J., Ritchie D.: Shapecoder: Discovering abstractions for visual programs from unstructured primitives. *arXiv preprint arXiv:2305.05661* (2023). Presented at SIGGRAPH 2023. URL: https://arxiv.org/abs/2305.05661, doi:10.48550/arXiv.2305.05661. 10

[JHC*21] Jones B., Hildreth D., Chen D., Baran I., Kim V. G., Schulz A.: Automate: A dataset and learning approach for automatic mating of cad assemblies. *ACM Transactions on Graphics (TOG) 40* (December 2021), 1–18. doi:10.1145/3478513.3480562. 3

[JNK*23] Jones B., Noeckel J., Kodnongbua M., Baran I., Schulz A.: B-rep matching for collaborating across cad systems. *ACM Transactions on Graphics 42* (August 2023). doi:10.1145/3592125. 3

[JWR22] Jones R. K., Walke H., Ritchie D.: Plad: Learning to infer shape programs with pseudo-labels and approximate distributions. *arXiv preprint arXiv:2011.13045* (2022). Presented at CVPR 2022. URL: https://arxiv.org/abs/2011.13045, doi:10.48550/arXiv.2011.13045. 10

[KMP*18] Kalyan A., Mohta A., Polozov O., Batra D., Jain P., Gulwani S.: Neural-guided deductive search for real-time program synthesis from examples. *arXiv* (April 2018). Published in ICLR 2018, International Conference on Learning Representations. doi:10.48550/arXiv.1804.01186. 3

[KSA23] Kuznetsov P., Spitsyn A., Arutyunov R.: Simplification of 3d cad model in voxel form for mechanical parts using a gan-based network. *Computer-Aided Design 162* (2023). URL: https://www.sciencedirect.com/science/article/pii/S0010448523001094, doi:10.1016/j.cad.2023.103461. 6

[LB25] Liu C., Bessmeltsev M.: State-of-the-art report in sketch processing. *Computer Graphics Forum* (2025). 2

[LCLT08] Liu J., Cao L., Li Z., Tang X.: Plane-based optimization for 3d object reconstruction from single line drawings. *IEEE Transaction on Pattern Analysis Machine Intelligence 30*, 2 (2008), 315–327. 3

[LCP*24] Liu Y., Chen J., Pan S., Cohen-Or D., Zhang H., Huang H.: Split-and-fit: Learning b-reps via structure-aware voronoi partitioning. *ACM Transactions on Graphics (TOG) 43*, 4 (2024), 108:1–108:13. URL: https://doi.org/10.1145/3658155, doi:10.1145/3658155. 6

[LGG*17] Lin T.-Y., Goyal P., Girshick R., He K., Dollár P.: Focal loss for dense object detection. *arXiv preprint arXiv:1708.02002* (2017). Presented at ICCV 2017. URL: https://arxiv.org/abs/1708.02002, doi:10.48550/arXiv.1708.02002. 5

[LOWS23] Liu Y., Obukhov A., Wegner J. D., Schindler K.: Point2cad: Reverse engineering cad models from 3d point clouds. *CVPR* (December 2023). doi:10.48550/arXiv.2312.04962. 3

[LPBM20] Li C., Pan H., Bousseau A., Mitra N. J.: Sketch2cad: Sequential cad modeling by sketching in context. *TOG 39*, 6 (Nov 2020). doi:10.1145/3414685.3417807. 1, 2, 3, 4

[LPBM22] Li C., Pan H., Bousseau A., Mitra N. J.: Free2cad: Parsing freehand drawings into cad commands. *TOG 41*, 4 (July 2022). doi:10.1145/3528223.3530133. 1, 2, 3, 4, 9, 14, 17

[LS96] Lipson H., Shpitalni M.: Optimization-based reconstruction of a 3d object from a single freehand line drawing. *Computer-Aided Design 28*, 8 (1996). 3

[LWJ*22] Lambourne J. G., Willis K., Jayaraman P. K., Zhang L., Sanghi A., Malekshan K. R.: Reconstructing editable prismatic cad from rounded voxel models. In *SIGGRAPH Asia Conference Papers* (2022). 3

[NGDGA*16] Nishida G., Garcia-Dorado I., G. Aliaga D., Benes B., Bousseau A.: Interactive sketching of urban procedural models. *ACM Transactions on Graphics (Proc. SIGGRAPH)* (2016). 3

[PCV16] Plumed R., Company P., Varley P. A.: Detecting mirror symmetry in single-view wireframe sketches of polyhedral shapes. *Computers & Graphics 59* (2016), 1–12. 3

[PMKB23] Puhachov I., Martens C., Kry P. G., Bessmeltsev M.: Reconstruction of machine-made shapes from bitmap sketches. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia) 42*, 6 (2023). 3

[RDM*24] Rukhovich D., Dupont E., Mallis D., Cherenkova K., Kacem A., Aouada D.: Cad-recode: Reverse engineering cad code from point clouds. *arXiv preprint arXiv:2412.14042* (2024). arXiv:2412.14042, doi:10.48550/arXiv.2412.14042. 3

[RGJ*23] Ritchie D., Guerrero P., Jones R. K., Mitra N. J., Schulz A., Willis K. D. D., Wu J.: Neurosymbolic models for computer graphics. *Computer Graphics Forum 42*, 2 (2023), 545–568. 2

[SGL*18] Sharma G., Goyal R., Liu D., Kalogerakis E., Maji S.: Csgnet: Neural shape parser for constructive solid geometry. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2018). 3

[SHM*16] Silver D., Huang A., Maddison C. J., Guez A., Sifre L., Driessche G. V. D., Schrittwieser J., Antonoglou I., Panneershelvam V., Lanctot M., Dieleman S., Grewe D., Nham J., Kalchbrenner N., Sutskever I., Lillicrap T., Leach M., Kavukcuoglu K., Graepel T., Hassabis D.: Mastering the game of go with deep neural networks and tree search. *Nature 529*, 7587 (2016), 484–489. URL: https://www.nature.com/articles/nature16961, doi:10.1038/nature16961. 6

[SKSK09] Schmidt R., Khan A., Singh K., Kurtenbach G.: Analytic drawing of 3d scaffolds. In *ACM transactions on graphics (Proc. SIGGRAPH Asia)* (2009), vol. 28. 2, 3

[SLK*20] Sharma G., Liu D., Kalogerakis E., Maji S., Chaudhuri S., Měch R.: Parsenet: A parametric surface fitting network for 3d point clouds. In *Proc. European Conference on Computer Vision (ECCV)* (2020). 3

[SLX*25] Sun Y., Li J., Xu Z., Zhang J., Liu X., Zhang D., Lu G.: Sketch2seq: Reconstruct CAD models from feature-based sketch segmentation. *IEEE Transactions on Visualization and Computer Graphics 31*, 10 (Oct. 2025), 8214–8230. doi:10.1109/TVCG.2025.3566544. 1, 2, 3, 4, 17

[SSS*17] Silver D., Schrittwieser J., Simonyan K., Antonoglou I., Huang A., Guez A., Hubert T., Baker L., Lai M., Bolton A., Chen Y., Lillicrap T., Hui F., Sifre L., Driessche G. V. D., Graepel T., Hassabis D.: Mastering the game of go without human knowledge. *Nature 550*, 7676 (2017), 354–359. URL: https://www.nature.com/articles/nature24270, doi:10.1038/nature24270. 6

[Sto08] Storer I.: Reflecting on professional practice : capturing an industrial designer's expertise to support the development of the sketching capabilities of novices. *Design and Technology Education: An International Journal 10*, 1 (May 2008), 54–72. 1

[TLS*19] Tian Y., Luo A., Sun X., Ellis K., Freeman W. T., Tenenbaum J. B., Wu J.: Learning to infer and execute 3d shape programs. *arXiv* (January 2019). Presented at ICLR 2019. doi:10.48550/arXiv.1901.02875. 3

[UyCS*22] Uy M. A., yu Chang Y., Sung M., Goel P., Lambourne J., Birdal T., Guibas L.: Point2cyl: Reverse engineering 3d objects from point clouds to extrusion cylinders. *CVPR* (June 2022). doi:10.48550/arXiv.2112.09329. 6

[WB25] Wei J., Bousseau A. (Eds.):. *A Blender Add-on for 3D Concept Sketching* (2025), ACM/EG Expressive Symposium - Posters and Demos. URL: http://www-sop.inria.fr/reves/Basilic/2025/WB25. 2, 9, 17, 18

[WJC*22] Willis K. D., Jayaraman P. K., Chu H., Tian Y., Li Y., Grandi D., Sanghi A., Tran L., Lambourne J. G., Solar-Lezama A., Matusik W.: Joinable: Learning bottom-up assembly of parametric cad joints. *CVPR* (April 2022). doi:10.48550/arXiv.2111.12772. 3

[WXW18] Wu Q., Xu K., Wang J.: Constructing 3d csg models from 3d raw point clouds. *Computer Graphics Forum 37*, 5 (2018). 3

[WXZ21] Wu R., Xiao C., Zheng C.: Deepcad: A deep generative network for computer-aided design models. *ICCV* (October 2021). doi:10.48550/arXiv.2105.09492. 3

[WZW*24] WANG H., ZHAO M., WANG Y., QUAN W., YAN D.-M.: VQ-CAD: Computer-aided design model generation with vector quantized diffusion. *Computer Aided Geometric Design 111* (2024), 102327. URL: https://www.sciencedirect.com/science/article/pii/S016783962400061X, doi:10.1016/j.cagd.2024.102327. 3

[XCS*14] XU B., CHANG W., SHEFFER A., BOUSSEAU A., MCCRAE J., SINGH K.: True2form: 3d curve networks from 2d sketches via selective regularization. *ACM Transactions on Graphics (SIGGRAPH 2014 Papers) 33* (2014). doi:10.1145/2601097.2601204. 3

[XPC*21] XU X., PENG W., CHENG C.-Y., WILLIS K. D., RITCHIE D.: Inferring cad modeling sequences using zone graphs. *CVPR* (April 2021). doi:10.48550/arXiv.2104.03900. 3

[YDSG21] YU X., DIVERDI S., SHARMA A., GINGOLD Y.: ScaffoldSketch: Accurate industrial design drawing in vr. In *Proceedings of ACM Symposium on User Interface Software and Technology* (2021), UIST. 2

[YLT13] YANG L., LIU J., TANG X.: Complex 3d general object reconstruction from line drawings. In *IEEE Int. Conference on Computer Vision* (2013). 3

[YZF*21] YANG L., ZHUANG J., FU H., WEI X., ZHOU K., ZHENG Y.: Sketchgnn: Semantic sketch segmentation with graph neural networks. *ACM Transactions on Graphics 40*, 3 (2021). 4

[ZHFL23] ZONG Z., HE F., FAN R., LIU Y.: P2cadnet: An end-to-end reconstruction network for parametric 3d cad model from point clouds. *CoRR abs/2310.02638* (2023). URL: https://arxiv.org/abs/2310.02638. 6

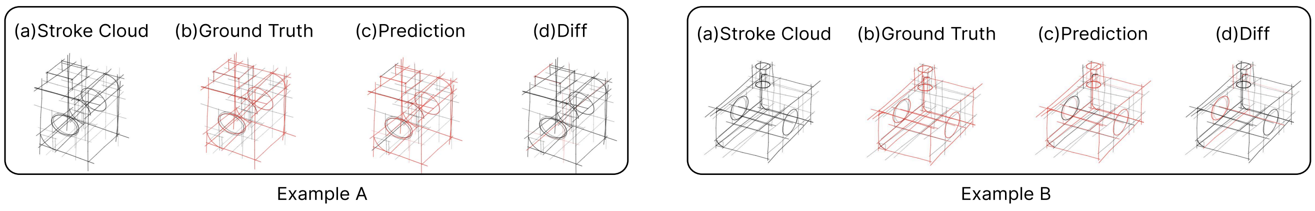


Figure 11: *Stroke filtering as a preprocessing step. We show two sets of results where strokes are classified as either used in the generation history or not. In both exampkes, some essential strokes are excluded, making the correct reconstructing infeasible.*

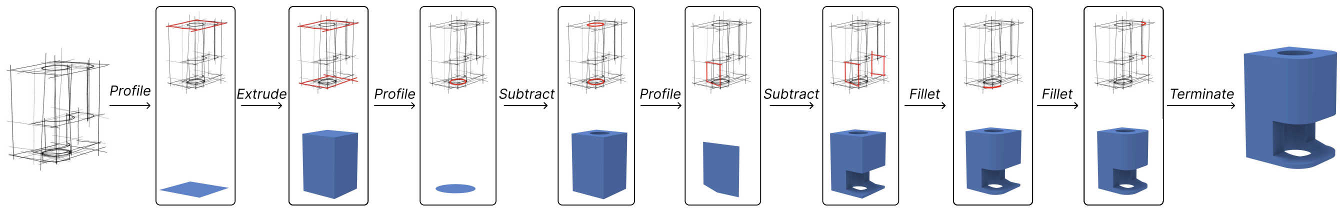


Figure 12: *We show the entire process of generating a CAD program from Dataset A. For each step, the selected strokes (highlighted in red) are shown at the top of the box, while the generated B-rep is shown at the bottom.*

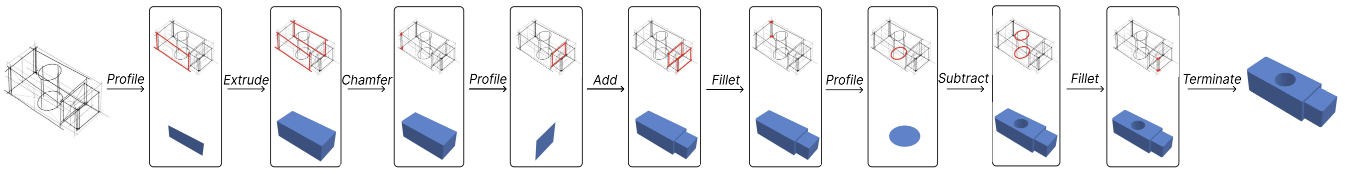


Figure 13: *We show the entire process of generating a CAD program from Dataset B. For each step, the selected strokes (highlighted in red) are shown at the top of the box, while the generated B-rep is shown at the bottom.*

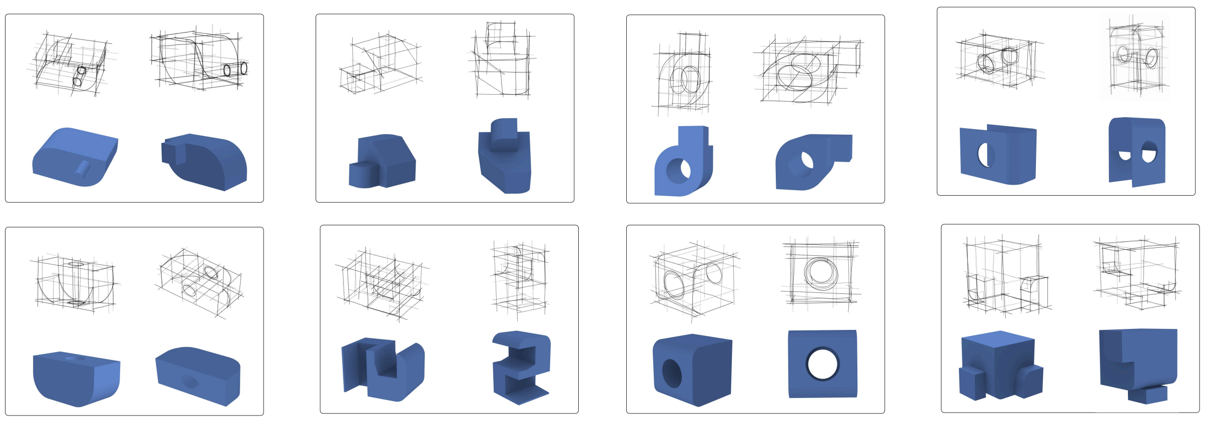


Figure 14: *We show eight results of CAD program generation from Dataset A. Each box contains one result, with the shape shown from two different perspectives.*

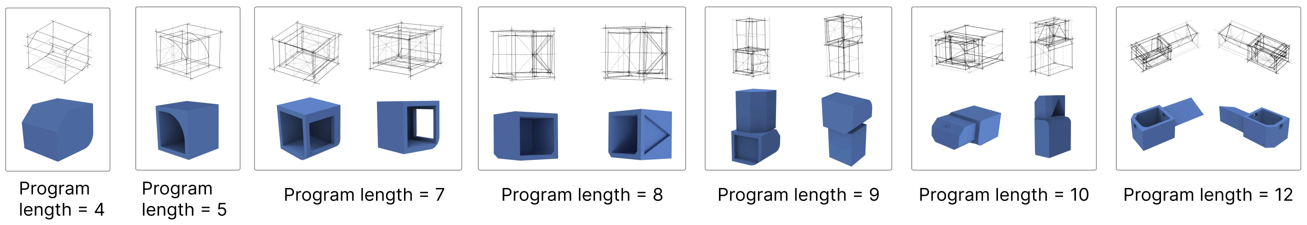


Figure 15: *We present seven results of CAD program generation from Dataset B with varying program length. Each box contains one result, with the shape shown from one or two different perspectives.*

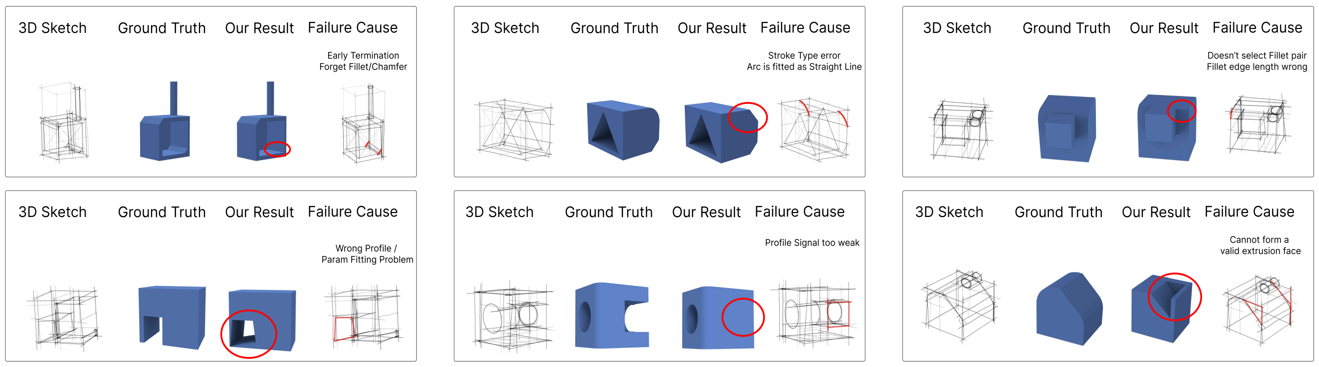

3D Sketch
Ground Truth
Our Result
Failure Cause
Early Termination Forget Fillet/Chamfer
Stroke Type error Arc is fitted as Straight Line
Doesn't select Fillet pair Fillet edge length wrong
Wrong Profile / Param Fitting Problem
Profile Signal too weak
Cannot form a valid extrusion face

Figure 16: *We present six failure cases. In each box, we show the input 3D sketch, the ground truth, our generated result, and the incorrectly selected strokes in the 3D sketch that led to the failure. Differences between our result and the ground truth are highlighted for clarity. The most common failures involve misclassification of small features such as* `fillets` *or* `chamfers`*, as seen in the first row.*

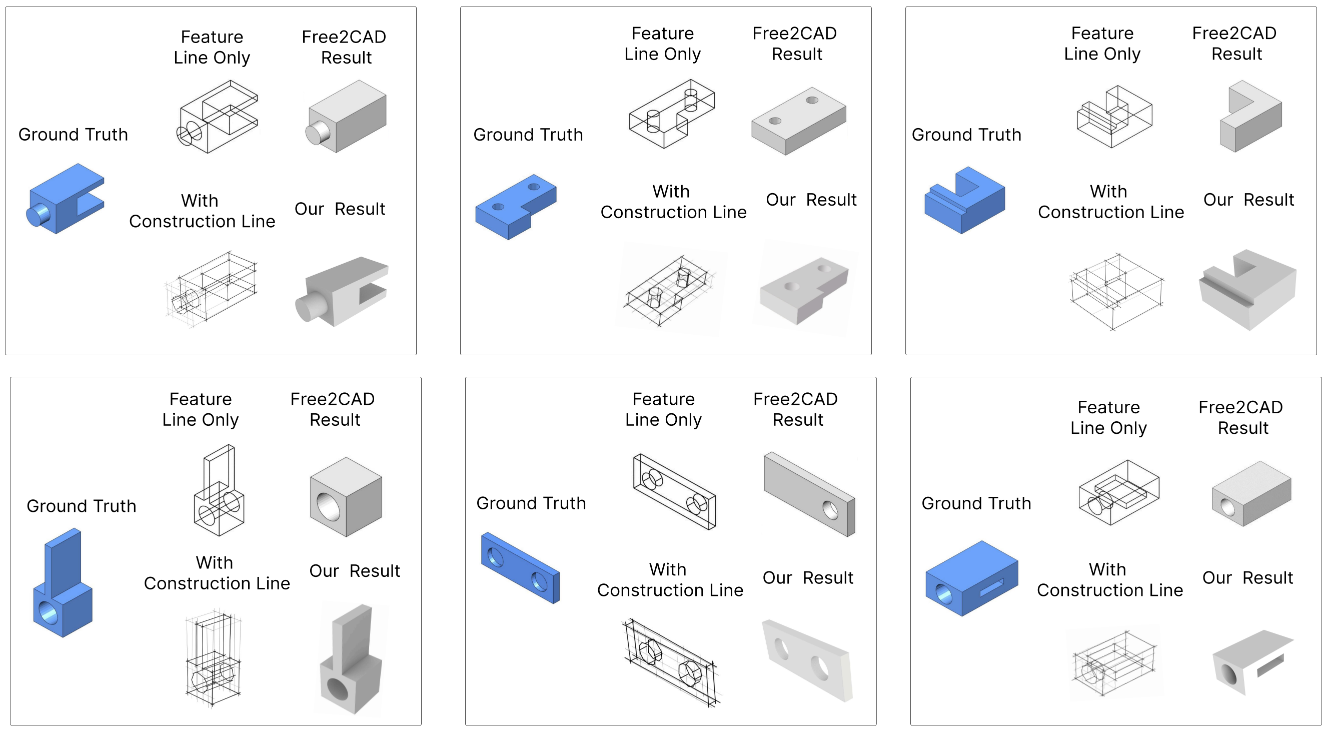


Figure 17: *We present six examples from Free2CAD [LPBM22]. In each box, the top row shows the ground truth shape, the input sketch for Free2CAD, and the result generated by their method. The bottom row shows our redrawn sketch with construction lines and the corresponding result produced by our system. Original figures copied from Free2CAD.*

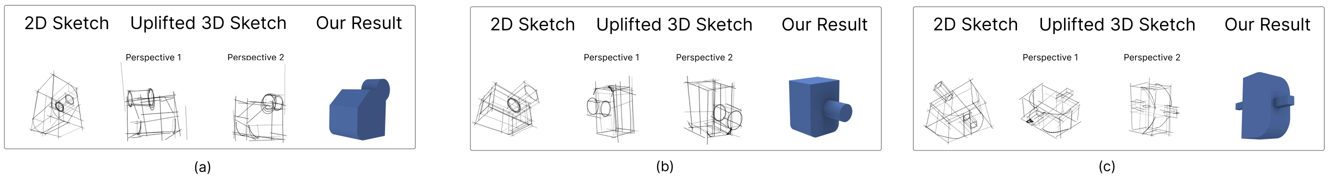


Figure 18: *We selected 2D sketches from a previous work [HLMB22], and then lift them back to 3D space using a previous method [HGSB22]. We show the resulting shapes. Although the lifting approach may introduce minor issues—as seen in (c), where the circle is distorted during the uplift process—our system can still make for valid interpretations based on the 3D sketch.*

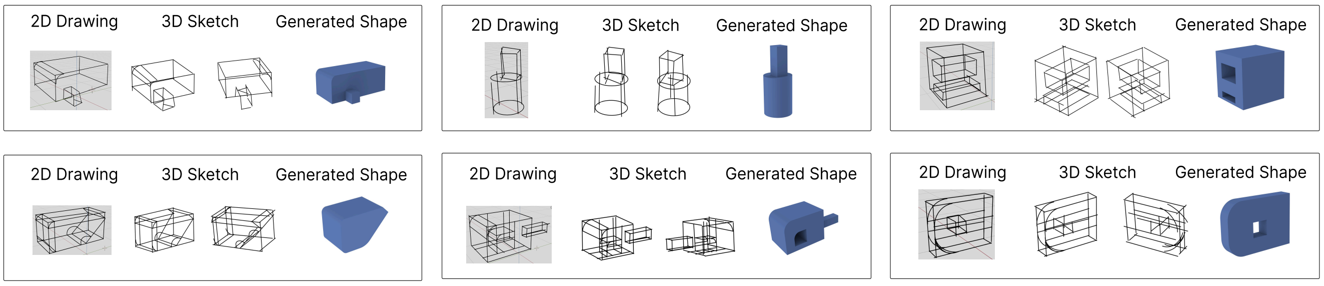


Figure 19: *We invited three students with limited CAD design experience and one student proficient in CAD design to use our system. The first row presents results from a non-proficient student, while the second row shows the work of the proficient student designer.*

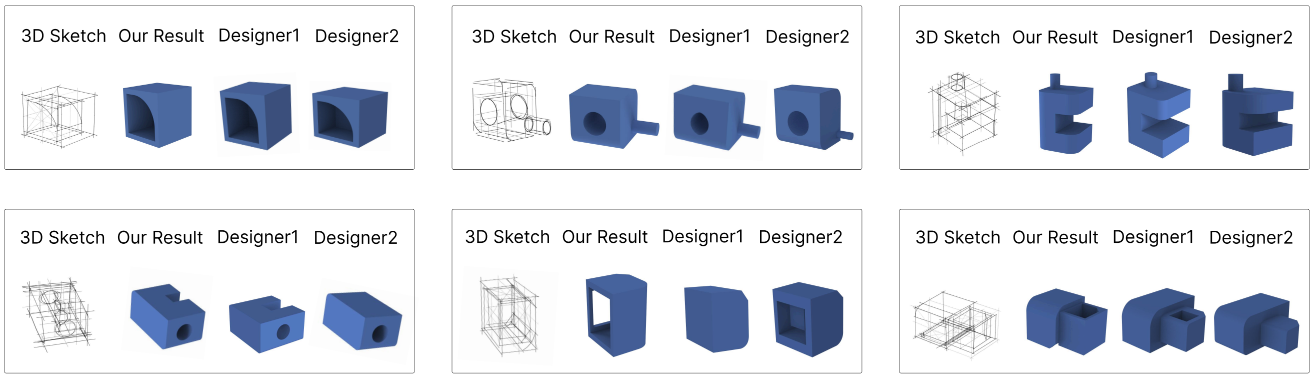


Figure 20: *Comparison between reconstructions by student designers and our method. Each student was given six 3D sketches and asked to recreate the corresponding B-rep shapes. The designers performed well on simpler 3D sketches (first row), but encountered difficulties with more complex ones (second row), where many lines appear cluttered especially for 3D sketches with multiple* `subtractions`*. In contrast, our method can still handle these cases.*

Appendix A: Graph Node Feature Representation

We show the details of our graph node features in Figure 21. Both stroke nodes and loop nodes contain 12 values.

For stroke nodes, the features include parametric information, opacity, circular attributes, stroke type encoding, and a binary label. There are five stroke types: straight lines, circular arcs, full circles, ellipses, and free-form curves, as shown in Table 5.

Table 5: *Node features for stroke nodes. Each stroke node has 12 values, including parametric and semantic features.*

Stroke Type	Parametric Function	Opacity	Circular Features	Stroke Type	Binary Label
Straight Line	Start and End points	Yes	/	1	0 or 1
Circular Arc	Start and End points	Yes	Center	2	0 or 1
Full Circle	Center and Normal	Yes	Radius + [0,0]	3	0 or 1
Ellipse	Center1 and Center2	Yes	Radius1, Radius2+ [0]	4	0 or 1
Free-form Curve	Start and End points	Yes	Sampled Point	5	0 or 1

For loop nodes, the first 11 values are padding, and the final value is a binary label indicating whether the loop is used. We do not assign additional features to loop nodes, as all necessary information can be derived from the strokes that form them.

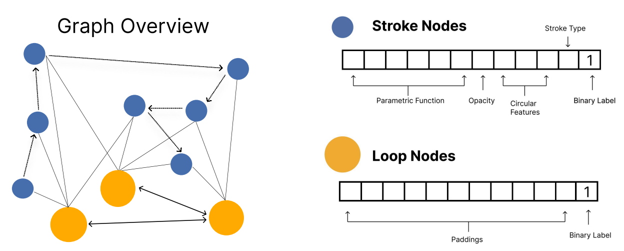


Figure 21: *Overview of graph node features. We represent each stroke node using 12 values that include parametric geometry, opacity, circular characteristics, stroke type encoding, and a binary label. Loop nodes only contain a binary label in the final feature slot, with the remaining dimensions zero-padded.*

Appendix B: Graph Encoder Architecture

We present our graph encoder in Figure 22. All tasks in our framework utilize this shared encoder to generate latent node embeddings for both stroke and loop nodes in the graph.

Each node in the input graph is initialized with a 12-dimensional feature vector. The output of the encoder is a 128-dimensional node embedding. The encoder first applies a graph convolutional layer to project the input features into a higher-dimensional space. This is followed by three residual blocks, each consisting of two graph convolutional layers with skip connections to preserve information flow. Finally, we apply another concluding graph convolutional layer is applied, followed by a ReLU activation to produce the final node embeddings.

Appendix C: Parameter Extraction for CAD Operations

Given the strokes (or loops) associated with each operation, we extract the continuous values required to parameterize them:

- **Profile:** A loop node is selected. We first project all strokes in the loop onto the best-fitting plane. Then, we extract one unique point from each stroke, resulting in n points from n strokes. Two points are considered identical if they lie within a threshold distance of $0.2 \times \max(\text{stroke\_length}_1, \text{stroke\_length}_2)$. These extracted points are used to define a plane by fitting with least squares:

$$\min_{\mathbf{n},d} \sum_{i=1}^{n} \left(\mathbf{n}^{\top}\mathbf{p}_i + d\right)^2 \qquad \|\mathbf{n}\| = 1$$

  where $\mathbf{p}_i$ are the extracted points and $\mathbf{n}$ is the plane normal.
- **Extrude:** A loop node is selected as the base face. The extrusion amount is computed as the Euclidean distance between the initial and final loop planes:

$$\theta_{\text{extrude}} = \|\mathbf{loop}_{\text{end}} - \mathbf{loop}_{\text{start}}\| .$$

- **Fillet:** The fillet radius is directly extracted from the selected arc stroke:

$$\theta_{\text{fillet}} = r_{\text{arc}} .$$

  To identify the corresponding B-rep edge, we find the edge equidistant to the two endpoints of the fillet stroke.
- **Chamfer:** The chamfer amount is approximated using the length of the selected edge:

$$\theta_{\text{chamfer}} = \frac{\|\mathbf{p}_{\text{end}} - \mathbf{p}_{\text{start}}\|}{\sqrt{2}},$$

  assuming a 45° chamfer angle. Similar to the fillet case, we locate the target B-rep edge as the one equidistant from the endpoints of the chamfer stroke.

Appendix D: Dataset Preparation using Monte Carlo Tree Search

To prepare the dataset for training our value function, we construct trees that explore all possible execution paths of the system. Since exhaustive enumeration is infeasible, we approximate this process using Monte Carlo Tree Search (MCTS), which prioritizes exploration along high-impact branches.

In our implementation, we first expand the tree until it reaches 100 leaf nodes, regardless of tree depth. Among these, we select the top 20 leaf nodes with the highest probabilities, as they contribute most significantly to the overall value. For the remaining nodes, we perform four random executions to estimate their value. In contrast, the top 20 nodes undergo full tree expansion to more accurately evaluate their final result. This hierarchical search strategy reduces the total number of branches while retaining the fidelity of value estimation.

Empirically, we observe that programs with 8 operations typically result in 300–500 tree states, while programs with 12 operations yield approximately 1200–1800 states.

Appendix E: Algorithm to Simulating Human Drawings

We propose a novel method to perturb a clean 3D sketches in order to simulate human-like drawing variations (Figure 23). The input to our system is a set of polylines, each consisting of 10 sampled

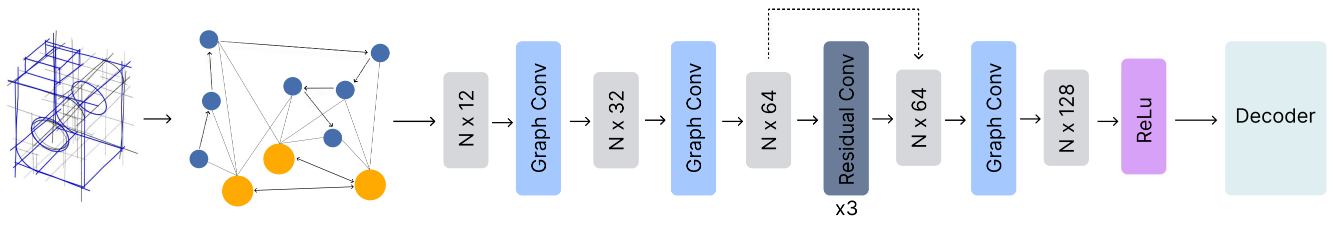


Figure 22: *Overview of the graph encoder architecture. The input is a heterogeneous graph G_t, where stroke and loop nodes are initialized with 12-dimensional features. The encoder applies graph convolutions to expand features to 128 dimensions through stacked layers and residual blocks, followed by a ReLU activation before passing to the decoder for task-specific predictions.*

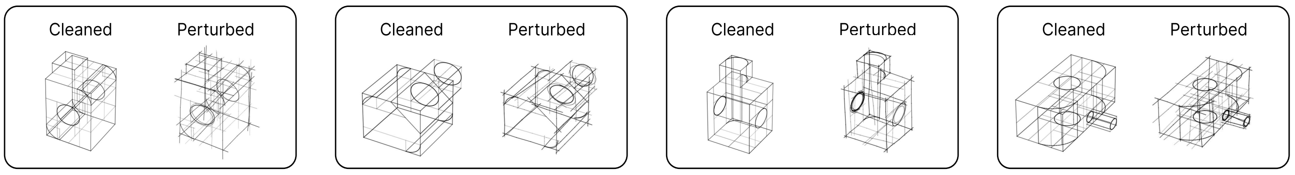


Figure 23: *3D Sketch perturbation to simulate human sketching. We show examples of clean and perturbed 3D Sketch. Perturbations are designed to emulate natural drawing variations such as jitter, overdrawing, stroke duplication, and deletion.*

points. The output has the same structure but with added perturbations.

Our perturbation process consists of two main steps. First, we perform stroke type fitting (as described in the graph construction section) to identify the type of each stroke. Second, we apply different perturbation strategies depending on the stroke type:

- **Straight Lines and Free-form Curves:** We simulate overdrawn strokes by randomly extending both the start and end points by a small fraction of the stroke length. We then perturb these endpoints in arbitrary directions, again by a fraction of the stroke length. Each intermediate point along the stroke is also randomly displaced by a small amount relative to the stroke length.
- **Arcs:** We first compute the arc's parametric representation. We then interpolate between the arc and a straight line connecting its endpoints, randomly blending the two 3D lines. The start and end points are also perturbed in arbitrary directions based on the stroke length.
- **Full Circles and Ellipses:** We treat full circles as a special case of ellipses where the two foci coincide and the major and minor radii are equal. We perturb these by randomly displacing the center based on the radius, and randomly modifying the radius itself. Additionally, we may repeat the entire stroke with high probability to simulate overdrawing.

After applying per-stroke perturbations, we further perturb the entire 3D Sketch by randomly removing 5% of the strokes and duplicating 10% of them.

Appendix F: UI Interface : Taking 2D Sketch as Input

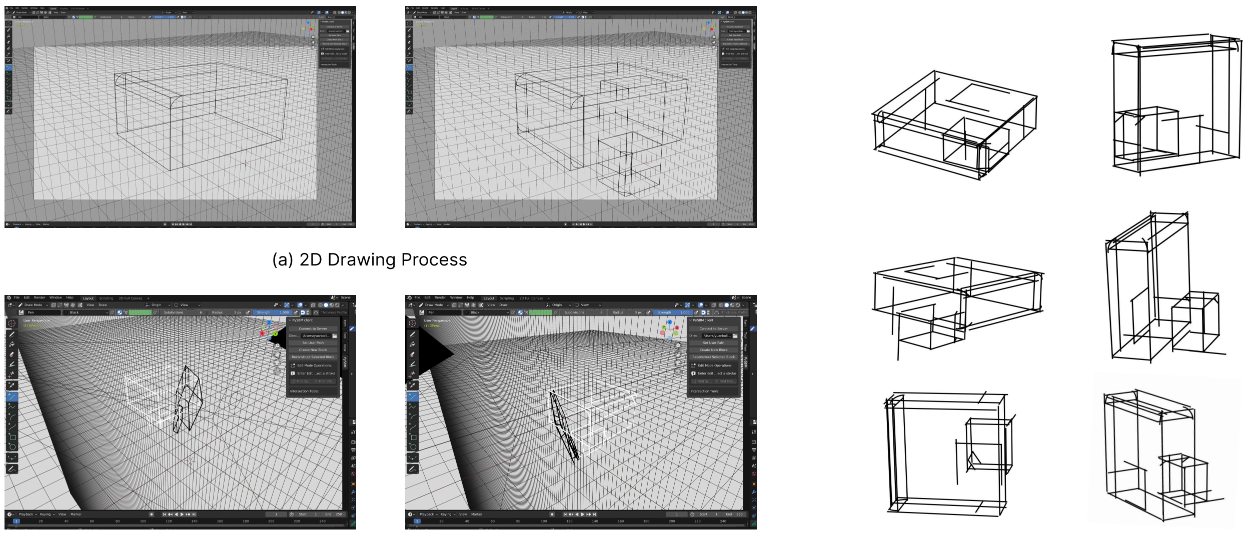
We build on the UI system introduced in prior work [WB25], which incorporates a 2D sketch lifting algorithm [HGSB22]. We chose this system due to our familiarity with it, though our method also supports other forms of input that produce 3D sketches. The system provides a Blender add-on that allows users to create 2D drawings directly within the software, which are then automatically lifted into 3D sketches. A demonstration is shown in Figure 24, with additional details available in the original work [WB25].

Appendix G: Participants' Usage of Construction Lines

We also observed that the participants exhibited diverse sketching habits, particularly in their use of construction lines.

Intermediate Lines All participants, both proficient and non-proficient in design, draw intermediate lines, as they found it easier to outline basic shapes first and then refine them, rather than attempting to draw the final feature lines directly (Figure 25). This observation aligns with our assumption that intermediate construction lines are frequently used in human drawings and play a crucial role in accurately recovering shapes. In contrast, prior works [LPBM22, SLX*25] does not account for such lines.

Grid Lines and Projection Lines Participants demonstrated different approaches to using additional construction lines, such as grid lines and projection lines, which do not appear in the intermediate shapes but assist in correcting perspectives. Overall, the student designer was more inclined to draw deliberate construction lines for perspective correction, whereas the other participants were less consistent. For instance, one participant never used projection or grid lines, noting that the built-in UI grid was sufficient for maintaining alignment. Others, however, found projection lines helpful when aligning distant features. Additionally, one participant observed that drawing larger shapes with long strokes made the interface difficult to manipulate, as correcting perspective in such

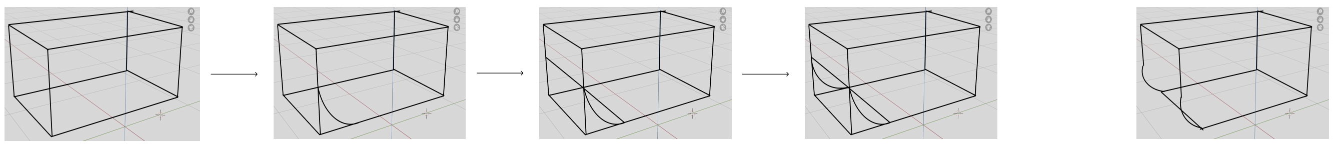


(a) 2D Drawing Process

(b) Lifted Strokes in Blender UI

(c) 3D Sketch Generated

Figure 24: *(a) illustrates the creation of a 2D sketch in Blender. (b) shows the corresponding lifted 3D sketch (in white) alongside the original 2D sketch (in black) within the user interface. (c) presents the final result of the lifted sketches in 3D space. Further details are provided in the original work [WB25].*



(a) Our User's Drawing Process (with intermediate shapes)

(b) Sketch with only feature lines

Figure 25: *We present an example of a participant's drawing in (a). The participant first sketched the entire cuboid, then added a curve to indicate the fillet operation. Then the user use projection lines to connect the edges of the cuboid. These project lines help the user to maintain alignment between the two fillet curves. In contrast, (b) shows the same shape drawn with only the feature lines, which is not how people typically sketch.*

cases was particularly challenging. While the use of construction lines provided some assistance, the process as a whole remained cumbersome.