

# Engineering Sketch Generation for Computer-Aided Design

Karl D.D. Willis Pradeep Kumar Jayaraman Joseph G. Lambourne Hang Chu Yewen Pu  
Autodesk Research

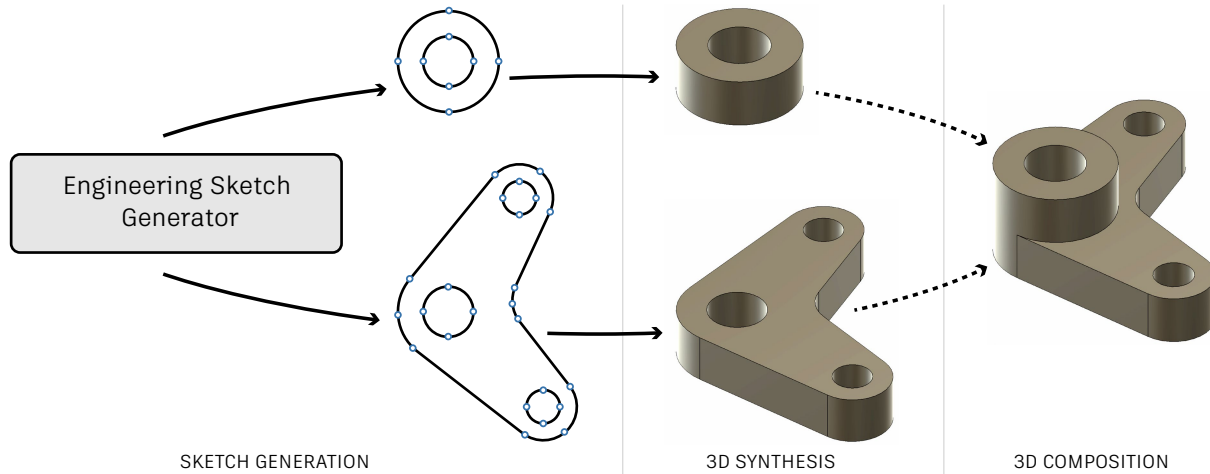


Figure 1: We tackle the problem of learning based engineering sketch generation as a first step towards synthesis and composition of solid models with an editable parametric CAD history.

## Abstract

Engineering sketches form the 2D basis of parametric Computer-Aided Design (CAD), the foremost modeling paradigm for manufactured objects. In this paper we tackle the problem of learning based engineering sketch generation as a first step towards synthesis and composition of parametric CAD models. We propose two generative models, *CurveGen* and *TurtleGen*, for engineering sketch generation. Both models generate curve primitives without the need for a sketch constraint solver and explicitly consider topology for downstream use with constraints and 3D CAD modeling operations. We find in our perceptual evaluation using human subjects that both *CurveGen* and *TurtleGen* produce more realistic engineering sketches when compared with the current state-of-the-art for engineering sketch generation.

## 1. Introduction

Parametric Computer-Aided Design (CAD) is the foremost 3D modeling paradigm used to design manufactured objects from automobile parts, to electronic devices, to furniture. Engineering sketches form the 2D basis of parametric

CAD, and refer specifically to composite curves made up of 2D geometric primitives (e.g. lines, arcs, circles), topological information about how the primitives connect together, and constraints defined using topology (e.g. coincidence, tangency, symmetry). Engineering sketches can be extruded or revolved to generate simple 3D solid bodies (Figure 1), and in turn combined using Boolean operations to build up complex shapes [22]. This workflow is common to all parametric CAD software and supported by all the major solid modeling kernels. Consequently, the ability to generate high quality engineering sketches is a major enabling technology for the automatic generation of solid models with an editable parametric CAD history.

Engineering sketch generation can be applied in a number of CAD workflows. For example, a long sought-after goal is the ability to automatically reverse engineer a parametric CAD model from noisy 3D scan data [1]. One way to realize this goal is to generate 2D CAD sketches from sparse scan data, just like human designers would, and apply suitable modeling operations to reconstruct the 3D CAD model. Engineering sketch generation can also be applied to auto-completion of user input. The ability to infer repetitive commands based on visual or geometric input could significantly ease user burden when producing complex engineering sketches. Another sought-after capabil-

ity is the generation of engineering sketches from approximate geometry like free-hand drawings. Often referred to as *beautification*, a generative model for engineering sketches could potentially improve user workflows over traditional approaches [9].

Despite recent advances with 2D vector graphic generation using data-driven approaches [21, 3, 26], there exists limited research on synthesizing engineering sketches directly. This is a challenging problem because engineering sketches contain disparate 2D geometric primitives coupled with topological information on how these primitives are connected together. The topology information is critical to ensure: 1) geometric primitives can be grouped into closed profile loops and lifted to 3D with modeling operations, and 2) constraints can be correctly defined using the topology, for example, two lines can be constrained to intersect at  $90^\circ$  if their endpoints are known to coincide. With the availability of large-scale engineering sketch and parametric CAD datasets [27, 35], we believe a data-driven approach that learns a generative model is a promising avenue to explore.

In this paper, we propose two generative models, **CurveGen** and **TurtleGen**, for the task of engineering sketch generation. CurveGen is an adaptation of PolyGen [23], where Transformer networks are used to predict the sketches autoregressively. While PolyGen generates polygonal mesh vertices and indexed face sets, CurveGen generates vertices lying on individual curves and indexed hyperedge sets that group the vertices into different curve primitives (line, arc, circle). This approach generates not only the individual curve geometry, but also the topology linking different curves, a vital requirement of engineering sketches. TurtleGen is an extension of TurtleGraphics [11], where an autoregressive neural network generates a sequence of *pen\_down*, *pen\_draw*, *pen\_up* commands, creating a series of closed loops which forms the engineering sketch. We find in our perceptual evaluation using human subjects that both CurveGen and TurtleGen produce more realistic engineering sketches when compared with the current state-of-the-art for engineering sketch generation. Importantly, both CurveGen and TurtleGen can generate geometry and topology directly without the expense of using a sketch constraint solver. This paper makes the following contributions:

- We introduce two generative models which tackle the problem of engineering sketch generation without the use of a sketch constraint solver.
- We use a novel sketch representation with our CurveGen model that implicitly encodes the sketch primitive type based on hyperedge cardinality.
- We show quantitative and qualitative results for engineering sketch generation, including the results of a perceptual study comparing our generative models with a state-of-the-art baseline.

## 2. Related Work

With the advent of deep learning a vast body of work has focused on generation of novel raster images. By contrast, significantly fewer works have tackled image generation using parametric curves such as lines, arcs, or Bézier curves. Parametric curves are widely used in 2D applications including vector graphics, technical drawings, floor plans, and engineering sketches—the focus of this paper. In this section we review work related to engineering sketch generation and its application to the synthesis and composition of solid CAD models.

**Vector Graphics** Vector graphics are used extensively in commercial software to enable the resolution independent design of fonts, logos, animations, and illustrations. Although a rich body of work has focused on *freeform* sketches [36, 37], we narrow our focus to structured vector graphics that consider shape topology, connectivity, or hierarchy relevant to engineering sketches.

Fonts are described with curves that form closed loop profiles and can contain interior holes to represent letters of different genus (e.g. *b* or *B*). SVG-VAE [21] is the first work to learn a latent representation of fonts using a sequential generative model for vector graphic output. DeepSVG [3] considers the structured nature of both fonts and icons with a hierarchical Transformer architecture. BézierGAN [4] synthesizes smooth curves using a generative adversarial network [12] and applies it to 2D airfoil profiles. More recently, Im2Vec [26] leverages differentiable rendering [20] for vector graphic generation trained only on raster images.

Common in prior research is the use of Bézier curves. Line, arc, and circle primitives are preferred in engineering sketches as they are easier to control parametrically, less prone to numerical issues (e.g. when computing intersections, offsetting, etc.) and can be lifted to 3D prismatic shapes like planes and cylinders rather than NURBS surfaces. In contrast to prior work, we pursue engineering sketch generation with line, arc, and circle primitives and explicitly consider topology for downstream application of constraints and use with 3D CAD modeling operations.

**Technical Drawings and Layout** Technical drawings take the form of 2D projections of 3D CAD models, often with important details marked by dimensions, notes, or section views. Han *et al.* [13] present the SPARE3D dataset, a collection of technical drawings generated from 3D objects with three axis-aligned and one isometric projection. The dataset is primarily aimed at spatial reasoning tasks. Pu *et al.* [25] show how freehand 2D sketches can be used to assist in the retrieval of 3D models from large object databases. Their system allows designers to make freehand sketches approximating the engineering sketches

which would be used in the construction of a 3D model. Egjazarian *et al.* [5] address the long standing problem of image vectorization for technical drawings. They predict the parameters of lines and Bézier curves with a Transformer based network, then refine them using optimization. In contrast, our system focuses on the generation of new sketch geometry suitable for use with 3D CAD modeling operations.

An emerging area of research considers learning based approaches to generative layout for graphic design [18, 38, 17] and floor plans [24, 15]. Although a different domain than engineering sketches, these works output high level primitives (e.g. rooms in a floor plan) that must be correctly connected to the overall layout and obey relevant constraints (e.g. number of bedrooms). Layout problems have similarities to engineering sketches where connectivity between parts of the sketch and relationships, such as parallel, symmetric, or perpendicular curves, are critical when designing.

**Program Synthesis** Another potential approach to engineering sketch generation is program synthesis. Recent work in this domain leverages neural networks in combination with techniques from the programming language literature to generate or infer programs capable of representing 2D [7, 8, 28] or 3D [28, 32, 6, 16] geometry. One can view engineering sketch generation as a program synthesis task where the geometry is represented as a sequence of programmatic commands that draw vertices and curves, constructing the sketch one piece at a time. Future applications of program synthesis would allow us to model more complex operations with programmatic constructs, such as repetitions (loops), symmetries (constraints), and modifications (refactoring and debugging).

**Reverse engineering** Another important field is reverse engineering 3D models from scan data or triangle meshes [2]. Typically 2D poly-linear profiles are generated by intersecting the mesh data with a plane, and then line and arc primitives can be least squares fitted in a way which respects a series of constraints [1]. Generative models which can be conditioned on approximate data provide an approach towards automating this part of the reverse engineering procedure. Recent learning-based approaches have tackled the challenge of reverse engineering parametric curves [10, 34], paving the way for reconstruction of trimmed surface models [19, 30, 31, 29]. In contrast to these works, we focus on 2D sketch generation as a building block towards the synthesis and composition of solid models, with parametric CAD history, using common modeling operations.

**Engineering Sketches** Most closely related to our work is SketchGraphs [27], a recently released dataset and baseline generative model for the task of engineering sketch gen-

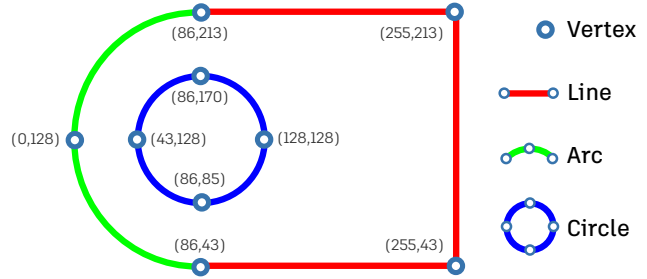


Figure 2: A hypergraph representation of a sketch consisting of three lines (2 vertices), an arc (3 vertices), and a circle (4 vertices).

eration. The SketchGraphs dataset consists of engineering sketches made by users of the Onshape CAD software. The SketchGraphs generative model works by predicting the underlying geometric constraint graph, and relies on a sketch constraint solver to determine the final configuration of the sketch. Unlike SketchGraphs we directly predict geometry and are not reliant on a sketch constraint solver. We observe qualitatively that our methods naturally generate geometry which conforms to the regular patterns seen in engineering sketches, such as horizontal and vertical lines and symmetries. The network can be considered to encode the constraint information *implicitly* in the geometric coordinates, allowing constrained sketches to be recovered in a post-processing step if required. We compare our work directly to SketchGraphs and present results that show we are able to produce more realistic engineering sketch output.

## 3. Method

### 3.1. Engineering Sketch Representation

To create an engineering sketch representation we consider a number of factors. 1) Engineering sketches are composed primarily from lines, arcs, and circles, while ellipses and splines are used less frequently [27, 35]. 2) Engineering sketches must obey constraints such as forming closed profiles, coinciding the end points, and forming 90 degree angles. 3) As the constraints remove many degrees of freedom, engineering sketches are both structured and sparse when compared to free-form sketches or vector graphics. We present two representations of engineering sketches that account for these design considerations: Sketch Hypergraph representation (used by CurveGen) and Turtle Graphics representation (used by TurtleGen).

**Sketch Hypergraph Representation** Under this representation, a sketch is represented as a hypergraph  $G = (V, E)$  where  $V = \{v_1, v_2, \dots, v_n\}$  denotes the set of  $n$  vertices, each vertex  $v_i = (x_i, y_i)$  consists of a vertex id  $i$  and its corresponding location  $x_i, y_i$ .  $E$  denotes a set of hy-

peredges that connects 2 or more vertices to form geometric primitives. The primitive curve type is implicitly defined by the cardinality of the hyperedge: line (2), arc (3), circle (4). Arcs are recovered by finding the circle that uniquely passes through the 3 vertices. Circles are recovered by a least squares fitting due to the over-parameterization by 4 points. Figure 2 shows the sketch hypergraph representation, consisting of 9 vertices and 5 hyperedges. In addition, the vertices are quantized to a  $256 \times 256$  grid to bias the generative model into generating few distinct coordinates by learning to produce repeated  $(x, y)$  values.

**Turtle Graphics Representation** The Turtle Graphics representation uses a *sequence* of drawing commands, which can be *executed* to form an engineering sketch in the hypergraph representation. Intuitively, the turtle representation can be thought of as a sequence of pen-up, pen-move actions, which iteratively *draws* the engineering sketch one loop at a time. It is used by Ellis *et al.* [8] to generate compositional and symmetric sketches. Specifically, the drawing commands of turtle graphics are specified by the following grammar:

$$\begin{aligned}
 \textit{Turtle} &\vdash [\textit{Loop}] \\
 \textit{Loop} &\vdash \textit{LoopStart} [\textit{Draw}] \\
 \textit{Draw} &\vdash \textit{Line} \mid \textit{Arc} \mid \textit{Circle} \\
 \textit{LoopStart} &\vdash \textit{loopstart}(\Delta) \\
 \textit{Line} &\vdash \textit{line}(\Delta) \\
 \textit{Arc} &\vdash \textit{arc}(\Delta, \Delta) \\
 \textit{Circle} &\vdash \textit{circle}(\Delta, \Delta, \Delta, \Delta) \\
 \Delta &\vdash (\textit{int}, \textit{int})
 \end{aligned}$$

Here, a *Turtle* program consists of a sequence of *Loops*, each loop consists of a start command *LoopStart*, followed by a sequence of [*Draw*] commands. The pen initially starts at  $(0, 0)$ . The *LoopStart* command starts a new loop by lifting the current pen, displacing/teleporting it by  $\Delta$  and putting it back down. The *Draw* command can be one of three primitives *Line*, *Arc*, and *Circle*, each parameterized by a different number of  $\Delta$  displacements, which extend the current loop without lifting the pen, displacing it *relative* to the current pen location. After a loop is completed, the pen returns/teleports back to  $(0, 0)$ . As with the hypergraph representation,  $\Delta$  values are quantized to a  $256 \times 256$  grid. The loops are ordered so that ones closest to  $(0, 0)$ , where distance is measured between the loop’s closest vertex to  $(0, 0)$ , are drawn first. We provide an example program in Section A.1 of the Supplementary Material.

### 3.2. Generative Models

We design and compare two different neural architectures on the task of engineering sketch generation: **CurveGen** and **TurtleGen**.

#### 3.2.1 CurveGen

CurveGen is our adaptation of the PolyGen [23] architecture applied to engineering sketch generation. CurveGen generates the sketch hypergraph representation *directly*. As with the original PolyGen implementation, we break the generation of  $G$  into two steps based on the chain rule: 1) generate the sketch vertices  $V$ , 2) generate the sketch hyperedges  $E$  conditioned on the vertices:

$$p(G) = \underbrace{p(E|V)}_{\text{Curve model}} \underbrace{p(V)}_{\text{Vertex model}},$$

where  $p(\cdot)$  are probability distributions. Figure 3 illustrates the two step generation process starting with the vertex model (left) and then the curve model (right). The final stage of recovering the curve primitives from the hyperedges is done as a post-process. We use the vertex model directly from PolyGen with 2D vertex coordinates and adapt our curve model from the PolyGen face model to work with 2D curves. We use 3 Transformer blocks in the vertex decoder, curve encoder, and curve decoder Transformer models [33]. Each block includes a multihead attention with 8 heads, layer normalization and a 2-layer multilayer perceptron (MLP) with 512D hidden, and 128D output dimensions. Dropout with rates 0.2 and 0.5 are applied in each block of the vertex and curve models, respectively, right after the MLP. The vertex and curve models are both trained by negative log likelihood loss against the ground truth data. Once the neural network is trained, we perform nucleus sampling [14] to directly generate samples in the hypergraph sketch representation. We refer the reader to Nash *et al.* [23] for further details.

#### 3.2.2 TurtleGen

The neural network for generating a program in the Turtle representation is a sequence generator. The sequence of turtle commands are encoded as a sequence of discrete valued tokens, where each of the commands *loopstart*, *line*, *arc*, *circle*, along with two tokens for sequence *start* and *end*, are represented as 1-hot vectors. The quantized integer coordinates are encoded as two 1-hot vectors for  $x$  and  $y$  each. For any given sketch in the hypergraph representation, we randomize the turtle sequence generation by randomly selecting the loop order, loop’s starting vertex, and the direction of drawing the loop. We discard long sequences over 100 turtle commands.

The neural network is a simple 9-layer Transformer with 512D hidden and 128D output dimensions. The network has seven linear branches of input and output, where the first branch corresponds to the type of command, and the remaining six branches correspond to three  $x$  and  $y$  coordinates. Commands with less than three points are padded



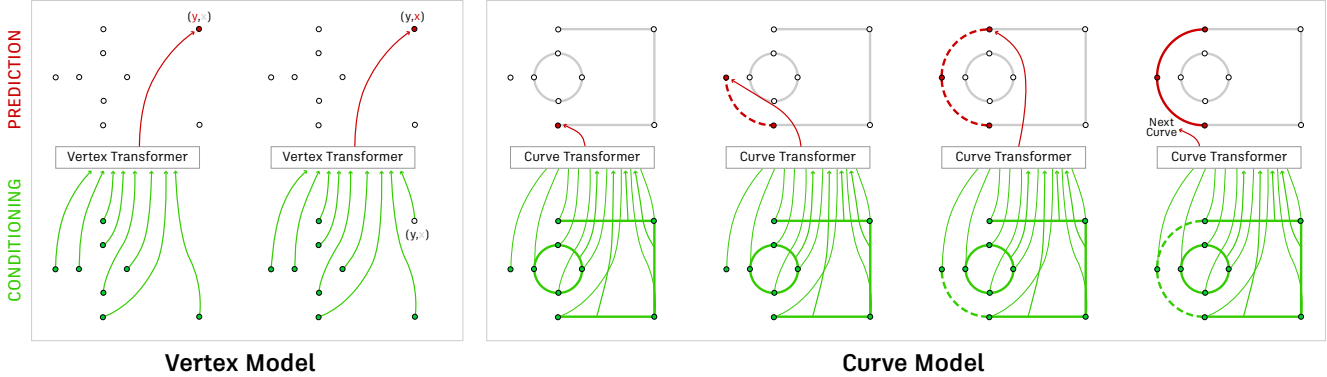


Figure 3: CurveGen, our variant of the PolyGen [23] architecture, first generates sketch vertices (left), and then generates sketch curves conditioned on the vertices (right).

with zeros. The input branches are concatenated and added with the conventional positional embedding after a linear layer, before they are fed into the Transformer network. The output branches are connected to the Transformer encoding at the previous sequence step. We store the first three ground-truth turtle steps from the training set as a fixed dictionary, which we randomly sample from to precondition the auto-regressive model at sampling time. Each sampled sequence is then executed to recover the sketch hypergraph representation, and define the geometry and topology.

## 4. Results

We now present quantitative and qualitative results on the task of engineering sketch generation, comparing the CurveGen and TurtleGen generative models with the SketchGraphs [27] generative model.

### 4.1. Data Preparation

We use the pre-filtered version of the SketchGraphs [27] dataset that contains 9.8 million engineering sketches with 16 or fewer curves. We remove duplicates from the dataset to promote data diversity during training, and ensure evaluation on unseen data at test time. We consider two sketches to be duplicates if they have identical topology and similar geometry. To detect similar geometry, sketches are uniformly scaled and the coordinates are quantized into a  $9 \times 9$  grid. Vertices are considered identical if they lie in the same grid square. The same quantization is also applied to the radii of circles and arcs. Lines are considered identical if the end points are identical. Arcs additionally check for an identical quantized radius. Circles check the center point and radius. We do not consider a sketch unique if both the topology and geometry match, but curve or vertex order is different. Using this approach we find that duplicates make up 87.01% of the SketchGraphs data. We remove all duplicates, invalid data (e.g. sketches with only points) and omit construction

curves. We use the official training split and after filtering have 1,106,328 / 39,407 / 39,147 sketches for the train, validation, and test sets respectively.

### 4.2. Experiment Setup

We train the CurveGen model on the SketchGraphs training set. Unlike the ShapeNet dataset used to train the original Polygen model, the SketchGraphs data does not have any notion of classes and represents a more challenging, but realistic scenario where human annotated labels are unavailable. We train for 2 million iterations with a batch size of 8 using 4 Nvidia V100 GPUs. We use the Adam optimizer with learning rate  $5e-4$ , and apply to the curve model data a jitter augmentation with the amount sampled from a truncated normal distribution with mean 0, variance 0.1 and truncated by the bounding box of the sketch vertices. Training time takes approximately 48 hours. We save the model with the lowest overall validation loss for evaluation.

We train the TurtleGen model on the same data. To ensure fair comparison, we train the model with a batch size of 128 for a total of 0.5 million iterations, which exposes the model to the same number of training data samples as CurveGen. Training is done with a single Nvidia V100 GPU and takes around 48 hours. We use the Adam optimizer with a learning rate of  $5e-4$ , as well as a learning rate scheduler that decreases the learning rate by a factor of 0.5 when the validation loss plateaus. Validation is conducted once every 500 training iterations. We save the model with the lowest overall validation loss for evaluation.

We train the SketchGraphs generative model using the official implementation with and without duplicates. We train for 150 epochs, as in the original paper, on a single Quadro RTX 6000 GPU. Training time takes approximately 27 hours. Following the advice of the SketchGraphs authors, we adjust the learning rate scheduler to reduce the learning rate at epochs 50 and 100. All other hyperparameters and settings follow the official implementation, in-

Table 1: Quantitative sketch generation results. *Bits per Vertex* and *Bits per Sketch* are the negative log-likelihood calculated over the test set; both are not directly comparable and reported separately for the vertex/curve models used in CurveGen. *Unique*, *Valid*, and *Novel* are calculated over 1000 generated sketches.

Model	Parameters	Bits per Vertex	Bits per Sketch	Unique %	Valid %	Novel %
CurveGen	<b>2,155,542</b>	1.75 / 0.20	176.69 / 30.64	<b>99.90</b>	<b>81.50</b>	<b>90.90</b>
TurtleGen	2,690,310	2.27	54.54	86.40	42.90	80.60
SketchGraphs	18,621,560	-	99.38	76.20	65.80	69.10
SketchGraphs (w/ Duplicates)	18,621,560	-	94.30	58.70	74.00	49.70

cluding the prediction of numerical node features. These provide improved geometry initialization before the data is passed to the OnShape constraint solver.

### 4.3. Quantitative Results

#### 4.3.1 Metrics

For quantitative evaluation we report the following metrics. **Bits per Vertex** is the negative log-likelihood of test examples averaged per-vertex and converted from nats to bits; lower is better. **Bits per Sketch** is the negative log-likelihood of test examples averaged per-sketch as a whole in bits; lower is better. For CurveGen we report the bits for both the vertex and curve models. **Unique** is the percentage of *unique* sketches generated within the sample set. We use the duplicate detection method described in Section 4.1 to find the percentage of unique sketches. A lower value indicates the model outputs more duplicate sketches. **Valid** is the percentage of *valid* sketches generated. Invalid sketches include curve fitting failures, curves generated with >4 vertices, or identical vertices within a curve. **Novel** is the percentage of *novel* sketches generated that are *not* identical to sketches in the training set. We again use the duplicate detection method described in Section 4.1. We evaluate the bits per vertex/sketch metrics on the withheld test set. All other metrics are evaluated on 1000 generated samples. We include non-*Valid* sketches, which often contain valid curves, when calculating the *Unique* and *Novel* metrics.

#### 4.3.2 Quantitative Comparison

Table 1 shows the results comparing CurveGen and TurtleGen with the SketchGraphs generative model. The parameter count for each model is provided for reference. Due to differences in sketch representation and terms in the negative log likelihood loss, the bits per vertex and bits per sketch results are not directly comparable between models, and only provided here for reference. For the SketchGraphs model, the prediction of numerical node features adds an additional term into the loss, giving a higher bits per sketch value than reported in the SketchGraphs paper. Invalid sketches occur most frequently with TurtleGen, where identical vertices within a curve are often predicted. Invalid

sketches from SketchGraphs are commonly due to arcs of near zero length. For the novel metric, it is reasonable to expect generative models to produce some identical sketches to those in the training data, such as simple circles, rectangles, and combinations thereof. The low percentage of novel sketches generated by SketchGraphs when trained on the dataset *with duplicates* suggests that the model memorizes sketches which are duplicated in the training data. Removing duplicates from the data helps improve variety in the output. For the remainder of the paper we report results from all models trained without duplicates.

### 4.4. Perceptual Evaluation

To understand how engineering sketches generated by each model compare to human designed sketches, we perform a perceptual evaluation using human subjects. In our two-alternative forced choice study, each participant is presented with one human designed and one generated engineering sketch and asked: “Which sketch is more realistic?”. Brief instructions are provided, including an illustration of an engineering sketch used in context, similar to Figure 1. We evaluate 1000 unique generative sketches from each model, with a consistent set of 1000 human designed sketches from the SketchGraphs test set. For each pair of sketches, we log the responses of three human subjects and

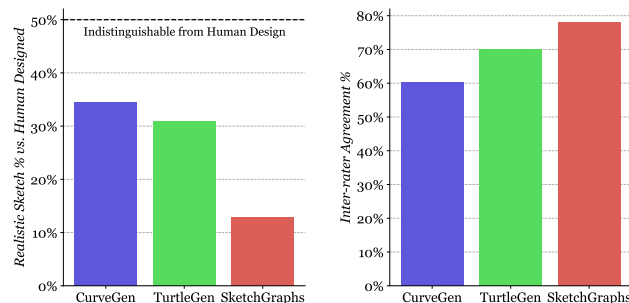


Figure 4: Results from our perceptual evaluation using human subjects to identify the most realistic engineering sketch. Left: The percentage of generated sketches classed as more realistic than human designed sketches. Right: The percentage of inter-rater agreement.

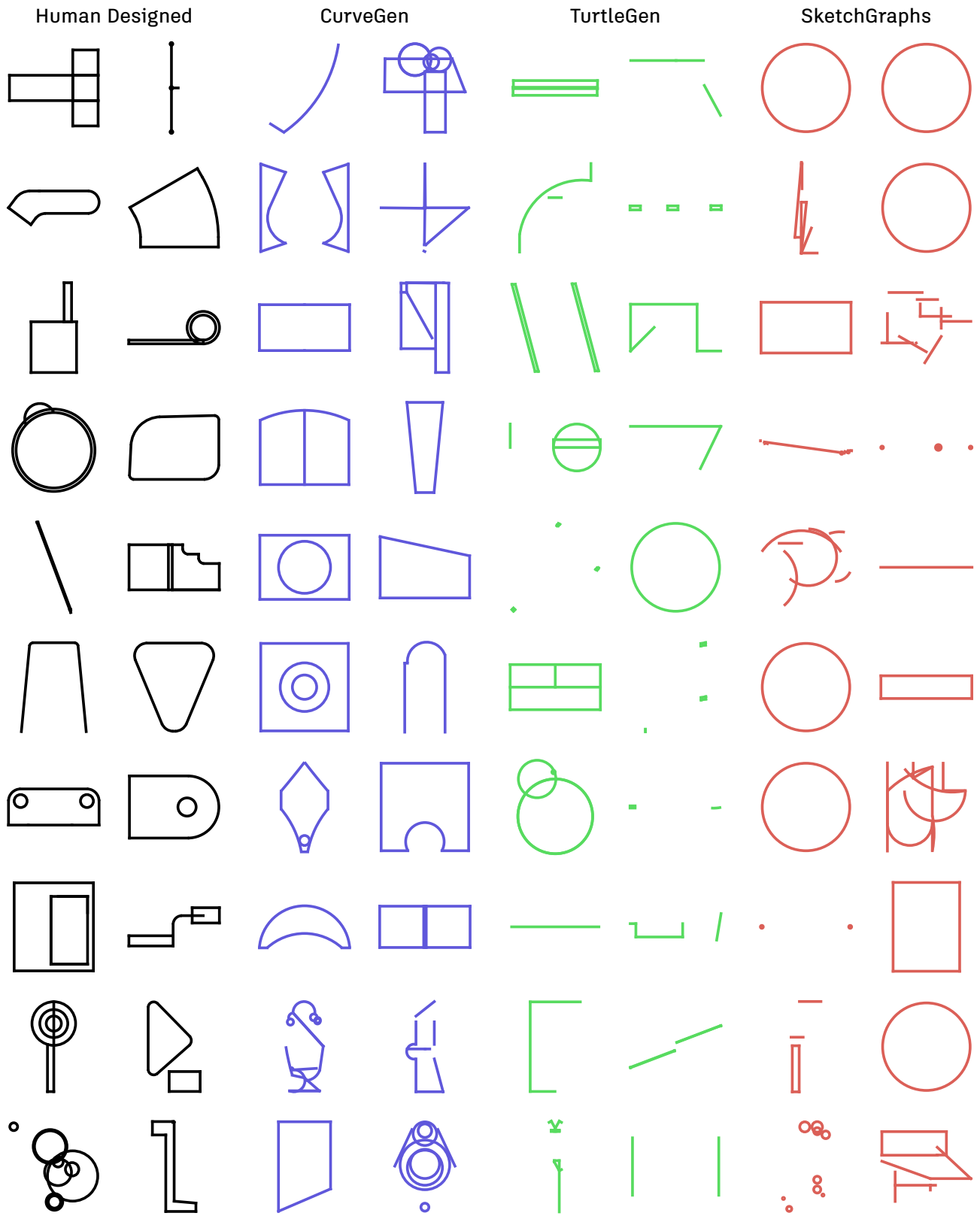


Figure 5: Qualitative sketch generation results. From left to right: human designed sketches from the SketchGraphs dataset, randomly selected sketches generated using the CurveGen, TurtleGen, and SketchGraphs generative models.

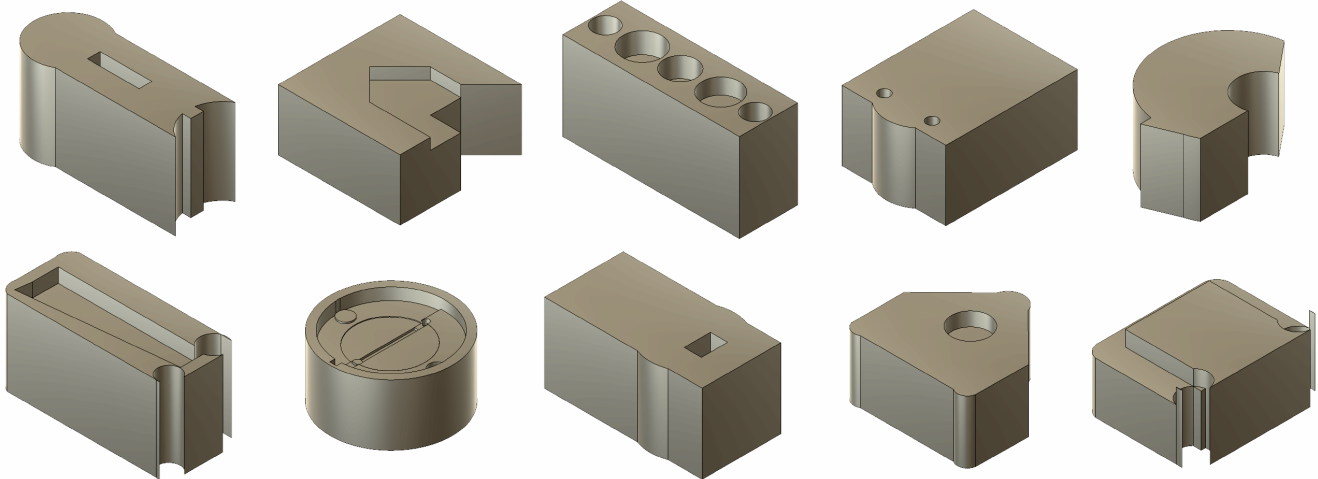


Figure 6: Examples of 3D geometry produced by extruding the closed profiles of sketches generated from CurveGen.

use the majority answer. We conduct the study using workers from Amazon Mechanical Turk. Figure 4, left shows the percentage of generated sketches classed as more realistic than human designed sketches; higher values are better. A value of 50% indicates the generated sketches are indistinguishable from human design. Figure 4, right shows the inter-rater agreement calculated as a percentage between each of the three human subjects. A lower value indicates there is more confusion between the generated and human designed sketches. The study results show that human subjects find CurveGen output to be the most realistic of the generated engineering sketches.

#### 4.5. Qualitative Results

Figure 5 shows human designed sketches from the SketchGraphs dataset beside randomly selected *valid* sketches generated using the CurveGen, TurtleGen, and

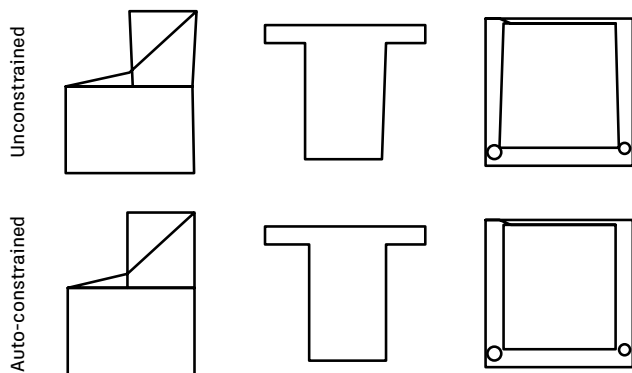


Figure 7: Comparison of sketches generated by CurveGen before (top) and after (bottom) applying automatic constraints in Autodesk AutoCAD.

SketchGraphs generative models. We observe that CurveGen in particular is able to consistently produce sketches with closed loops, symmetrical features, perpendicular lines, and parallel lines. We provide additional qualitative results in Section A.2 of the supplementary material.

**Sketch to Solid CAD Models** A key motivation of the current work is to enable the synthesis and composition of solid CAD models. In Figure 6 we demonstrate how engineering sketches generated by CurveGen with closed loop profiles can be lifted into 3D using the extrude modeling operation in a procedural manner.

**Sketch Constraints** The geometric output of our generative models can be post-processed to apply sketch constraints and build a constraint graph. Figure 7 shows how the auto-constrain functionality in Autodesk AutoCAD can enforce parallel and perpendicular lines within a given tolerance. The unconstrained output from CurveGen is shown on the top row, and has a number of lines that are close to perpendicular. The auto-constrained output on the bottom row snaps these lines to perpendicular and establishes constraints for further editing.

## 5. Conclusion

In this paper we presented the CurveGen and TurtleGen generative models for the task of engineering sketch generation and demonstrated that they produce more realistic output when compared with the current state-of-the-art. We believe engineering sketches are an important building block on the path to synthesis and composition of solid models with an editable parametric CAD history. Promising future directions include modeling higher-order constructs such as constraints and repetitions in the underlying design.



## References

- [1] Pál Benko, Géza Kós, Tamás Várady, László Andor, and Ralph Martin. Constrained fitting in reverse engineering. *Computer Aided Geometric Design*, 19(3):173–205, 2002.
- [2] Francesco Buonamici, Monica Carfagni, Rocco Furferi, Lapo Governi, Alessandro Lapini, and Yary Volpe. Reverse engineering modeling methods and tools: a survey. *Computer-Aided Design and Applications*, 15(3):443–464, 2018.
- [3] Alexandre Carlier, Martin Danelljan, Alexandre Alahi, and Radu Timofte. DeepSVG: A Hierarchical Generative Network for Vector Graphics Animation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
- [4] Wei Chen, Kevin Chiu, and Mark D Fuge. Airfoil Design Parameterization and Optimization Using Bézier Generative Adversarial Networks. *AIAA Journal*, 58(11):4723–4735, 2020.
- [5] Vage Egiazarian, Oleg Voynov, Alexey Artemov, Denis Volkhonskiy, Aleksandr Safin, Maria Taktasheva, Denis Zorin, and Evgeny Burnaev. Deep vectorization of technical drawings. In *European Conference on Computer Vision*, pages 582–598. Springer, 2020.
- [6] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 9169–9178, 2019.
- [7] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B Tenenbaum. Learning to infer graphics programs from hand-drawn images. *arXiv preprint arXiv:1707.09627*, 2017.
- [8] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv:2006.08381*, 2020.
- [9] Jakub Fišer, Paul Asente, Stephen Schiller, and Daniel Šykora. Advanced drawing beautification with shipshape. *Computers & Graphics*, 56:46–58, 2016.
- [10] Jun Gao, Chengcheng Tang, Vignesh Ganapathi-Subramanian, Jiahui Huang, Hao Su, and Leonidas J Guibas. Deep spline: Data-driven reconstruction of parametric curves and surfaces. *arXiv:1901.03781*, 2019.
- [11] Ron Goldman, Scott Schaefer, and Tao Ju. Turtle geometry in computer graphics and computer-aided design. *Computer-Aided Design*, 36(14):1471–1482, 2004.
- [12] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, page 2672–2680, 2014.
- [13] Wenyu Han, Siyuan Xiang, Chenhui Liu, Ruoyu Wang, and Chen Feng. Spare3d: A dataset for spatial reasoning on three-view line drawings. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14690–14699, 2020.
- [14] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *International Conference on Learning Representations (ICLR)*, 2019.
- [15] Ruizhen Hu, Zeyu Huang, Yuhan Tang, Oliver Van Kaick, Hao Zhang, and Hui Huang. Graph2plan: Learning floor-plan generation from layout graphs. *ACM Transactions on Graphics (TOG)*, 39(4):118–1, 2020.
- [16] Kacper Kania, Maciej Zięba, and Tomasz Kajdanowicz. Ucsq-net—unsupervised discovering of constructive solid geometry tree. *arXiv:2006.09102*, 2020.
- [17] Hsin-Ying Lee, Weilong Yang, Lu Jiang, Madison Le, Irfan Essa, Haifeng Gong, and Ming-Hsuan Yang. Neural design network: Graphic layout generation with constraints. *European Conference on Computer Vision (ECCV)*, 2020.
- [18] Jianan Li, Jimei Yang, Aaron Hertzmann, Jianming Zhang, and Tingfa Xu. Layoutgan: Generating graphic layouts with wireframe discriminators. In *International Conference on Learning Representations (ICLR)*, 2019.
- [19] Lingxiao Li, Minhyuk Sung, Anastasia Dubrovina, Li Yi, and Leonidas J Guibas. Supervised fitting of geometric primitives to 3d point clouds. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2652–2660, 2019.
- [20] Tzu-Mao Li, Michal Lukáč, Michaël Gharbi, and Jonathan Ragan-Kelley. Differentiable vector graphics rasterization for editing and learning. *ACM Transactions on Graphics (TOG)*, 39(6):1–15, 2020.
- [21] Raphael Gontijo Lopes, David Ha, Douglas Eck, and Jonathon Shlens. A learned representation for scalable vector graphics. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7930–7939, 2019.
- [22] H. Masuda. Topological operators and boolean operations for complex-based nonmanifold geometric models. *Computer-Aided Design*, 25(2):119–129, 1993.
- [23] Charlie Nash, Yaroslav Ganin, SM Ali Eslami, and Peter Battaglia. Polygen: An autoregressive generative model of 3d meshes. In *International Conference on Machine Learning (ICML)*, pages 7220–7229. PMLR, 2020.
- [24] Nelson Nauata, Kai-Hung Chang, Chin-Yi Cheng, Greg Mori, and Yasutaka Furukawa. House-gan: Relational generative adversarial networks for graph-constrained house layout generation. In *European Conference on Computer Vision (ECCV)*, pages 162–177. Springer, 2020.
- [25] Jiantao Pu, Kuiyang Lou, and Karthik Ramani. A 2d sketch-based user interface for 3d cad model retrieval. *Computer-aided Design - CAD*, 2, 01 2005.
- [26] Pradyumna Reddy, Michael Gharbi, Michal Lukac, and Niloy J Mitra. Im2vec: Synthesizing vector graphics without vector supervision. *arXiv:2102.02798*, 2021.
- [27] Ari Seff, Yaniv Ovadia, Wenda Zhou, and Ryan P Adams. Sketchgraphs: A large-scale dataset for modeling relational geometry in computer-aided design. *arXiv:2007.08506*, 2020.
- [28] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhansu Maji. Csgnet: Neural shape parser for constructive solid geometry. In *IEEE Conference*

- on *Computer Vision and Pattern Recognition (CVPR)*, pages 5515–5523, 2018.
- [29] Gopal Sharma, Difan Liu, Subhansu Maji, Evangelos Kalogerakis, Siddhartha Chaudhuri, and Radomír Měch. Parsenet: A parametric surface fitting network for 3d point clouds. In *European Conference on Computer Vision (ECCV)*, pages 261–276. Springer, 2020.
  - [30] Dmitriy Smirnov, Mikhail Bessmeltsev, and Justin Solomon. Learning manifold patch-based representations of man-made shapes. *arXiv:1906.12337*, 2019.
  - [31] Dmitriy Smirnov, Matthew Fisher, Vladimir G Kim, Richard Zhang, and Justin Solomon. Deep parametric shape predictions using distance fields. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 561–570, 2020.
  - [32] Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. Learning to infer and execute 3d shape programs. In *International Conference on Learning Representations (ICLR)*, 2019.
  - [33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, pages 5998–6008, 2017.
  - [34] Xiaogang Wang, Yuelang Xu, Kai Xu, Andrea Tagliasacchi, Bin Zhou, Ali Mahdavi-Amiri, and Hao Zhang. Pie-net: Parametric inference of point cloud edges. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
  - [35] Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. Fusion 360 gallery: A dataset and environment for programmatic cad reconstruction. *arXiv:2010.02392*, 2020.
  - [36] Peng Xu. Deep learning for free-hand sketch: A survey. *arXiv:2001.02600*, 2020.
  - [37] Chuan Yan, David Vanderhaeghe, and Yotam Gingold. A benchmark for rough sketch cleanup. *ACM Trans. Graph.*, 39(6), Nov. 2020.
  - [38] Xinru Zheng, Xiaotian Qiao, Ying Cao, and Rynson WH Lau. Content-aware generative modeling of graphic design layouts. *ACM Transactions on Graphics (TOG)*, 38(4):1–15, 2019.

## A. Supplementary Material

### A.1. TurtleGen Program Example

One possible TurtleGen program for the sketch in Figure 2 is as follows:

```
[  
  loopstart((86, 43)),  
  line((169, 0)),  
  line((0, 170)),  
  line((-169, 0)),  
  arc((-86, -85), (86, -85)),  
  loopstart((86, 85)),  
  circle((43, 43), (-43, 43), (-43, -43))  
]
```

The pen starts and returns to  $(0, 0)$  for each loop. The first five lines draw the outer loop, lifting the pen to  $(86, 43)$  and drawing counter clock-wise. The last two lines draw the inner circle, lifting the pen to  $(86, 85)$  vertex, and drawing counter clock-wise.

### A.2. Additional Qualitative Results

We provided additional qualitative sketch generation results in Figure 8 and Figure 9.

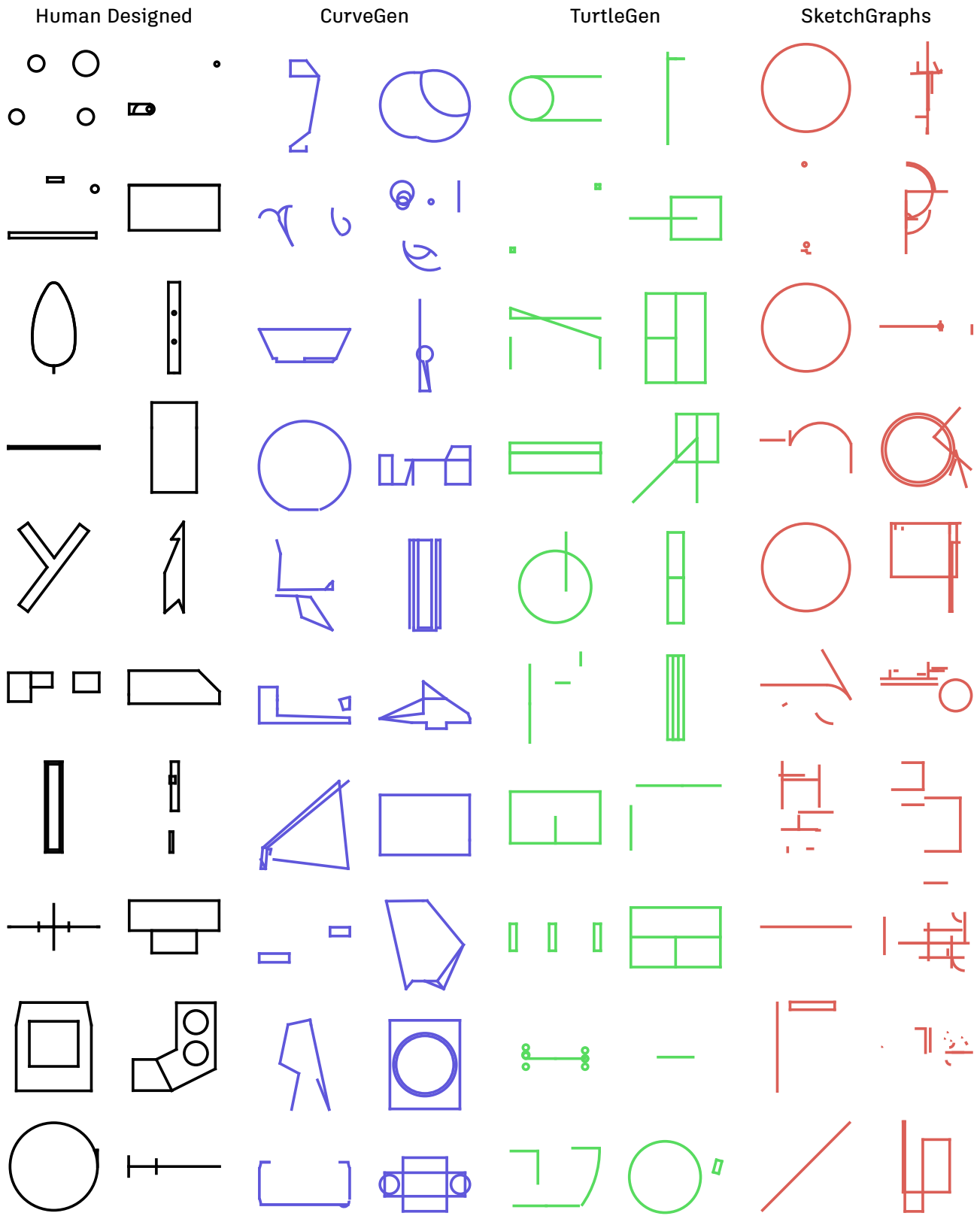


Figure 8: Additional qualitative sketch generation results. From left to right: human designed sketches from the SketchGraphs dataset, randomly selected sketches generated using the CurveGen, TurtleGen, and SketchGraphs generative models.

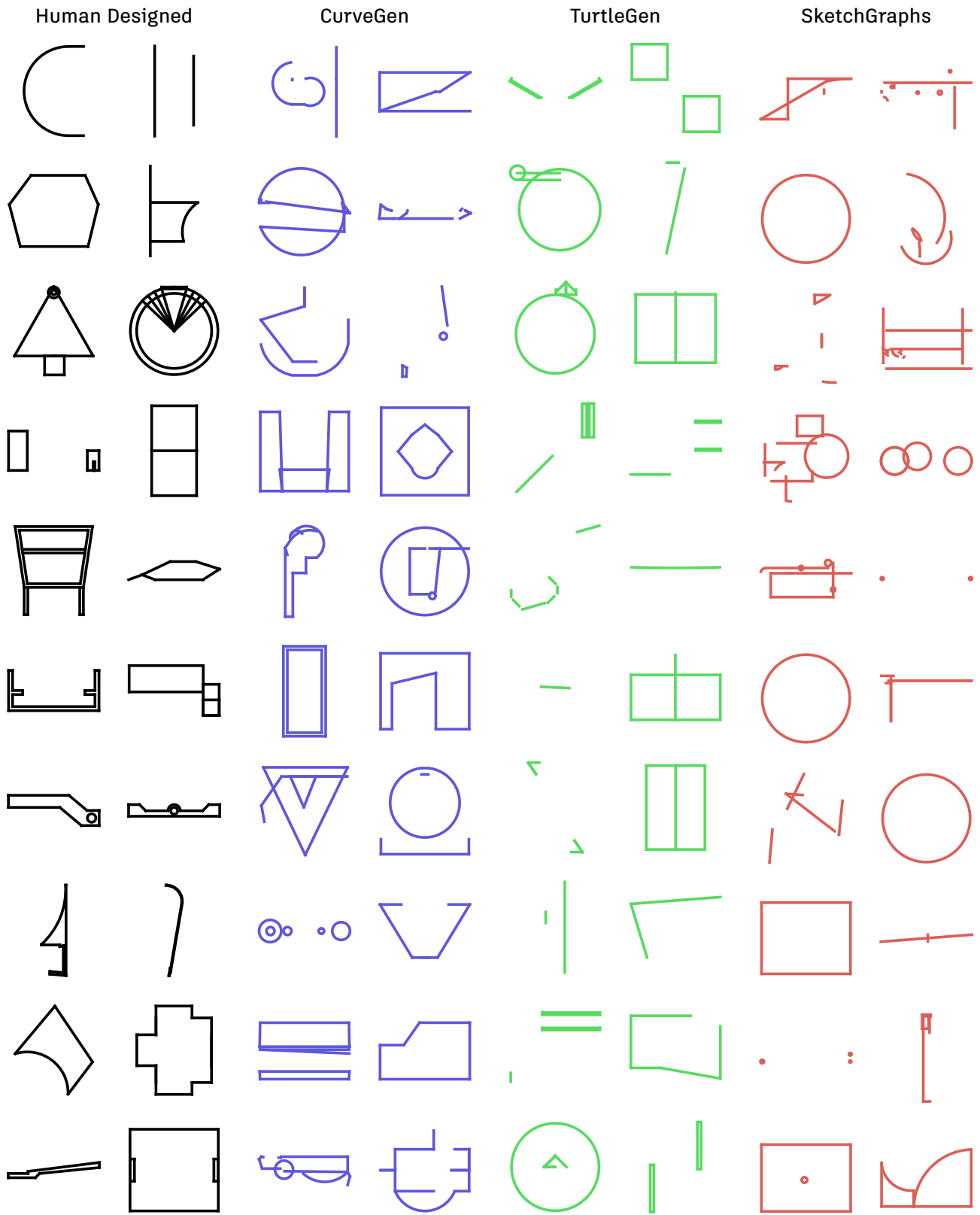


Figure 9: Additional qualitative sketch generation results. From left to right: human designed sketches from the SketchGraphs dataset, randomly selected sketches generated using the CurveGen, TurtleGen, and SketchGraphs generative models.