This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPAMI.2020.3044749, IEEE Transactions on Pattern Analysis and Machine Intelligence

Neural Shape Parsers for Constructive Solid Geometry

Gopal Sharma† Rishabh Goyal$^{\iota}$ Difan Liu† Evangelos Kalogerakis† Subhransu Maji†
University of Massachusetts, Amherst†, University of Illinois at Urbana-Champaign$^{\iota}$
{gopalsharma,dliu,kalo,smaji}@cs.umass.edu†, rgoyal6@illinois.edu$^{\iota}$

Abstract—Constructive Solid Geometry (CSG) is a geometric modeling technique that defines complex shapes by recursively applying boolean operations on primitives such as spheres and cylinders. We present CSGNET, a deep network architecture that takes as input a 2D or 3D shape and outputs a CSG program that models it. Parsing shapes into CSG programs is desirable as it yields a compact and interpretable generative model. However, the task is challenging since the space of primitives and their combinations can be prohibitively large. CSGNET uses a convolutional encoder and recurrent decoder based on deep networks to map shapes to modeling instructions in a feed-forward manner and is significantly faster than bottom-up approaches. We investigate two architectures for this task — a vanilla encoder (CNN) - decoder (RNN) and another architecture that augments the encoder with an explicit memory module based on the program execution stack. The stack augmentation improves the reconstruction quality of the generated shape and learning efficiency. Our approach is also more effective as a shape primitive detector compared to a state-of-the-art object detector. Finally, we demonstrate CSGNET can be trained on novel datasets without program annotations through policy gradient techniques.

Index Terms—Constructive Solid Geometry, Reinforcement Learning, Shape Parsing.

1 INTRODUCTION

In recent years, there has been a growing interest in generative models of 2D or 3D shapes, especially through the use of deep neural networks as image or shape priors. However, current methods are limited to the generation of low-level shape representations consisting of pixels, voxels, or points. Human designers, on the other hand, rarely model shapes as a collection of these individual elements. For example, in vector graphics modeling packages (*e.g.*, Inkscape, Illustrator, and so on), shapes are often created through higher-level primitives, such as parametric curves (*e.g.*, Bezier curves) or basic shapes (*e.g.*, circles, polygons), as well as operations acting on these primitives, such as boolean operations, deformations, extrusions, and so on. Describing shapes with higher-level primitives and operations is highly desirable for designers since it is compact and makes subsequent editing easier. It may also better capture certain aspects of human shape perception such as view invariance, compositionality, and symmetry [1].

The goal of our work is to develop an algorithm that parses shapes into their constituent modeling primitives and operations within the framework of Constructive Solid Geometry (CSG) [2]. CSG is a popular geometric modeling framework where shapes are generated by recursively applying boolean operations, such as union or intersection, on simple geometric primitives, such as spheres or cylinders. Figure 1 illustrates an example where a 2D shape (top) and a 3D shape (bottom) are generated as a sequence of operations over primitives or a *visual program*. Yet, parsing a shape into its CSG program poses a number of challenges. First, the number of primitives and operations is not the same for all shapes *i.e.*, our output does not have constant dimensionality, as in the case of pixel arrays, voxel grids, or fixed point sets. Second, the order of these instructions matter — small changes in the order of operations can significantly change the generated shape. Third, the number of possible programs grows exponentially with the program length, making learning and inference challenging.

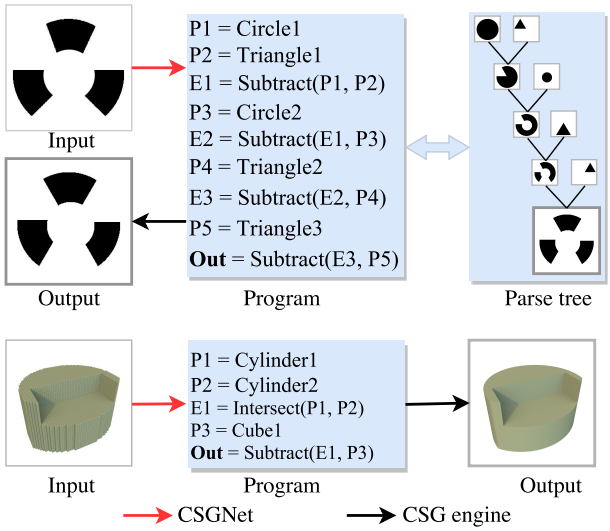


Fig. 1: **Our shape parser produces a program that generates an input 2D or 3D shape**. On top is an input image of 2D shape, its program and the underlying parse tree where primitives are combined with boolean operations. On the bottom is an input voxelized 3D shape, the induced program, and the resulting shape from its execution.

Existing approaches for CSG parsing are predominantly search-based. A significant portion of related literature has focused on approaches to efficiently estimate primitives in a bottom-up manner, and to search for their combinations using heuristic optimization. While these techniques can generate complex shapes, they are prone to noise in the input and are generally slow. Our contribution is a neural network architecture called CSGNET that

generates the program in a feed-forward manner. The approach is inspired by the ability of deep networks for generative sequence modeling such as for speech and language. As a result CSGNET is efficient at test time, as it can be viewed as an *amortized search* [3] procedure. Furthermore, it be used as an initialization for search-based approaches leading to improvements in accuracy at the cost of computation.

At a high-level, CSGNET is an encoder-decoder architecture that encodes the input shape using a convolutional network and decodes it into a sequence of instructions using a recurrent network (Figure 2). It is trained on a large synthetic dataset of automatically generated 2D and 3D programs (Table 2). However, this leads to poor generalization when applied to new domains. To adapt models to new domains without program annotations, we employ policy gradient techniques from the reinforcement learning literature [4]. Combining the parser with a CSG rendering engine allows the networks to receive feedback based on the visual difference between the input and generated shape, and the parser is trained to minimize this difference (Figure 2). Furthermore, we investigate two network architectures: a vanilla recurrent network (CSGNET), and a new variant called CSGNETSTACK (Figure 3). This new variant stores intermediate shapes produced during the execution of the CSG program, inspired by call or execution stacks [5]. This stack can also be seen as a form of explicit memory in our network encoding the intermediate program state. Our experiments demonstrate that this improves the overall accuracy of the generated programs while using less training data.

We evaluate the CSGNET and CSGNETSTACK architectures on a number of shape parsing tasks. Both offer consistently better performance than a nearest-neighbor baseline and are significantly more efficient than an optimization based approach. Reinforcement learning improves their performance when applying them to new domains without requiring ground-truth program annotations making the approach more practical (Table 4). We also investigate the effect of the training data size and reward choices used in the policy gradient algorithm [6] on the performance of the parser. Finally, we evaluate the performance on the task of primitive detection and compare it with a Faster R-CNN detector [7] trained on the same dataset. CSGNET offers 4.2% higher Mean Average Precision (MAP) and is $4\times$ faster compared to the Faster R-CNN detector, suggesting that joint reasoning about the presence and ordering of objects leads to better performance for object detection (Table 6).

This paper extends our work that first appeared in [8], adding to it an analysis on effect of reward shaping and training set size on the performance, as well as the stack-augmented network architecture. Our PyTorch [9] implementation is publicly available at: https://hippogriff.github.io/CSGNet/.

2 Related Work

CSG parsing has a long history and a number of approaches have been proposed in the literature over the past 20 years. Much of the earlier work can be categorized as "bottom-up" and focuses on the problem of converting a boundary representation (b-Rep) of the shape to a CSG program. Our work is more related to program generation approaches using neural networks which have recently seen a revival in the context of natural language, graphics, and visual reasoning tasks. We briefly summarize prior work below.

2.1 Bottom-up shape parsing

An early example of a grammar-based shape parsing approach is the "pictorial structure" model [10]. It uses a tree-structured grammar to represent articulated objects and has been applied to parsing and detecting humans and other categories [11]–[13]. However, the parse trees are often shallow and these methods rely on accurate bottom-up proposals to guide parsing (*e.g.*, face and upper-body detection for humans). In contrast, primitive detection for CSG parsing is challenging as shapes change significantly when boolean operations are applied to them. Approaches, such as [14]–[16], assume an exact boundary representation of primitives which is challenging to estimate from noisy or low-resolution shapes. This combined with the fact that parse trees for CSG can be significantly deeper makes bottom-up parsing error prone. Evolutionary approaches have also been investigated for optimizing CSG trees [17]–[19], however, they are computationally expensive.

Thus, recent work has focused on reducing the complexity of search. Tao *et al.* [20] directly operates on input meshes, and converts the mixed domain of CSG trees (discrete operations and continuous primitive locations) to a discrete domain that is suitable for boolean satisfiability (SAT) based program synthesizers. This is different from our approach which uses a neural network to generate programs without relying on an external optimizer.

2.2 Inverse procedural modeling

A popular approach to generate 3D shapes and scenes is to infer context-free, often probabilistic "shape grammars" from a small set of exemplars, then sample grammar derivations to create new shapes [21]–[24]. This approach called Inverse Procedural Modeling (IPM) has also been used in analysis-by-synthesis image parsing frameworks [25]–[27].

Recent approaches employ CNNs to infer parameters of objects [28] or whole scenes [29] to aid procedural modeling. A similar trend is observed in graphics applications where CNNs are used to map input images or partial shapes to procedural model parameters [30]–[32]. Wu *et al.* [33] detect objects in scenes by employing a network for producing object proposals and a network that predicts whether there is an object in a proposed segment, along with various object attributes. Eslami *et al.* [34] use a recurrent neural network to attend to one object at a time in a scene, and learn to use an appropriate number of inference steps to recover object counts, identities and poses.

Our goal is fundamentally different: given a generic grammar describing 2D or 3D modeling instructions and a target image or shape, our method infers a derivation, or more specifically a modeling program, that describes it. The underlying grammar for CSG is quite generic compared to specialized shape grammars. It can model shapes in several different classes and domains (*e.g.*, furniture, logos, *etc.*).

2.3 Neural program induction

Our approach is inspired by recent work in using neural networks to infer programs expressed in some high-level language, *e.g.*, to answer question involving complex arithmetic, logical, or semantic parsing operations [35]–[43]. Approaches, such as [44], [45], produce programs composed of functions that perform compositional reasoning on an image using an execution engine consisting of neural modules [46]. Similarly, our method produces a program

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPAMI.2020.3044749, IEEE Transactions on Pattern Analysis and Machine Intelligence

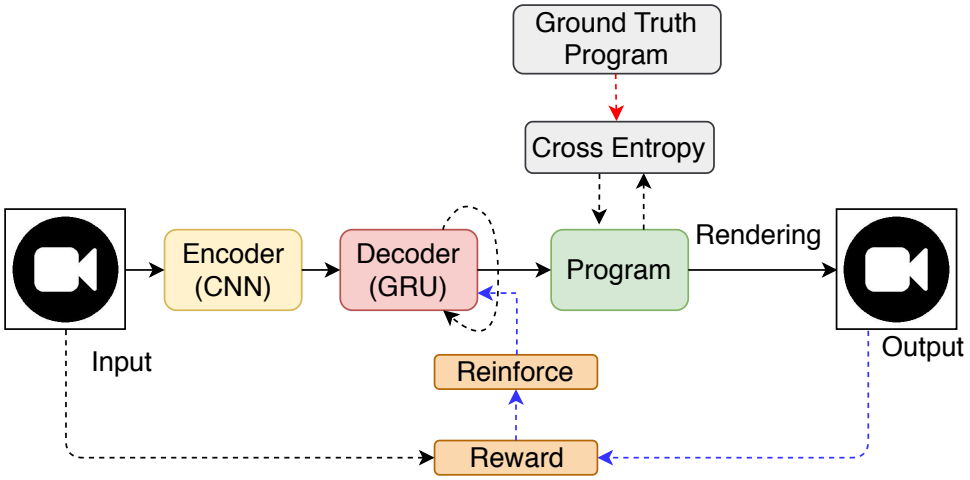


Fig. 2: **Overview of our approach.** Our neural shape parser consists of two parts: first at every time step encoder takes as input a target shape (2D or 3D) and outputs a feature vector through CNN. Second, a decoder maps these features to a sequence of modeling instructions yielding a visual program. The rendering engine processes the program and outputs the final shape. The training signal can either come from ground truth programs when such are available, or in the form of rewards after rendering the predicted programs.

consisting of shape modeling instructions to match a target image by incorporating a shape renderer.

Other related work include the recent work by Tian *et al.* [47], which proposes a program induction architecture for 3D shape modeling. Here programs contain a variety of primitives and symmetries are incorporated with loops. While this is effective for categories such as chairs, the lack of boolean operations is limiting. A more complex approach is that of Ellis *et al.* [48], who synthesize hand-drawn shapes by combining (lines, circles, rectangles) into Latex programs. Program synthesis is posed as a constraint satisfaction problem which is computationally expensive and can take hours to solve. In contrast, our feed-forward model that takes a fraction of a second to generate a program.

2.4 Primitive fitting

Deep networks have recently been applied to a wide range of primitive fitting problems for 2D and 3D shapes. Tulsiani *et al.* [49] proposed a volumetric CNN that predicts a fixed number of cuboidal primitives to describe an input 3D shape. Zou *et al.* [50] proposed an LSTM-based architecture to predict a variable number of boxes given input depth images. Li *et al.* [51] introduced a point cloud based primitive fitting network where shapes are represented as an union of primitives. Paschalidou *et al.* [52] uses superquadrics instead of traditional cuboids. Genova *et al.* [53] proposed a network that predicts local implicit functions decomposing the input shape into 3D Gaussian blobs. Huang *et al.* [54] decompose an image by detecting primitives and arranging them into layers. Gao *et al.* [55] train deep network to produce control points for splines using input images and point cloud. Recent networks such as BSP-Net [56] and CvxNet [57] are built on the concept of binary space partitioning to produce a collection of convexes that approximates the input point cloud or an image. Deprelle *et al.* [58] proposed representing shapes as the combination of learned deformable elementary 3D structures. The above approaches are trained to minimize reconstruction error like ours. On the other hand, they focus on predicting primitives, while our method also learns modeling operations (CSG) on them.

3 Designing a Neural Shape Parser

In this section, we first present a neural shape parser, called CSGNet, that induces programs based on a CSG grammar given only 2D/3D shapes as input. We also present another shape parser variant, called CSGNetStack, which incorporates a stack as a form of explicit memory and results in improved accuracy and faster training. We show that both variants can be trained to produce CSG programs in a supervised learning setting when ground-truth programs are available. When these are not available, we show that reinforcement learning can be used based on policy gradient and reward shaping techniques. Finally, we describe ways to improve the shape parsing at test time through a post-processing stage.

CSGNet. The goal of a **shape parser** π is to produce a sequence of instructions given an input shape. The parser can be implemented as an encoder-decoder using neural network modules as shown in Figure 2. The **encoder** takes as input an image I and produces an encoding $\Phi(I)$ using a CNN. The **decoder** Θ takes as input $\Phi(I)$ and produces a probability distribution over programs P represented as a sequence of instructions. Decoders can be implemented using Recurrent Neural Networks (RNNs). We employ Gated Recurrent Units (GRUs) [59] that have been widely used for sequence prediction tasks such as generating natural language and speech. The overall network can be written as $\pi(I) = \Theta \circ \Phi(I)$. We call this basic architecture as CSGNet (see also Figure 3, left).

CSGNetStack. The above architecture can further be improved by incorporating feedback from the renderer back to the network. More specifically, the encoder can be augmented with an execution stack that stores the result of the renderer at every time step along with the input shape. This enables the network to adapt to both current and previous rendered results. To accomplish this, our CSG rendering engine executes the program instructions produced by the decoder with the help of stack $S = \{s_t : t = 1, 2 \ldots\}$ at each time step t. The stack is updated after every instruction is executed and contains intermediate shapes produced by previous boolean operations or simply an initially drawn shape primitive. This stack

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPAMI.2020.3044749, IEEE Transactions on Pattern Analysis and Machine Intelligence

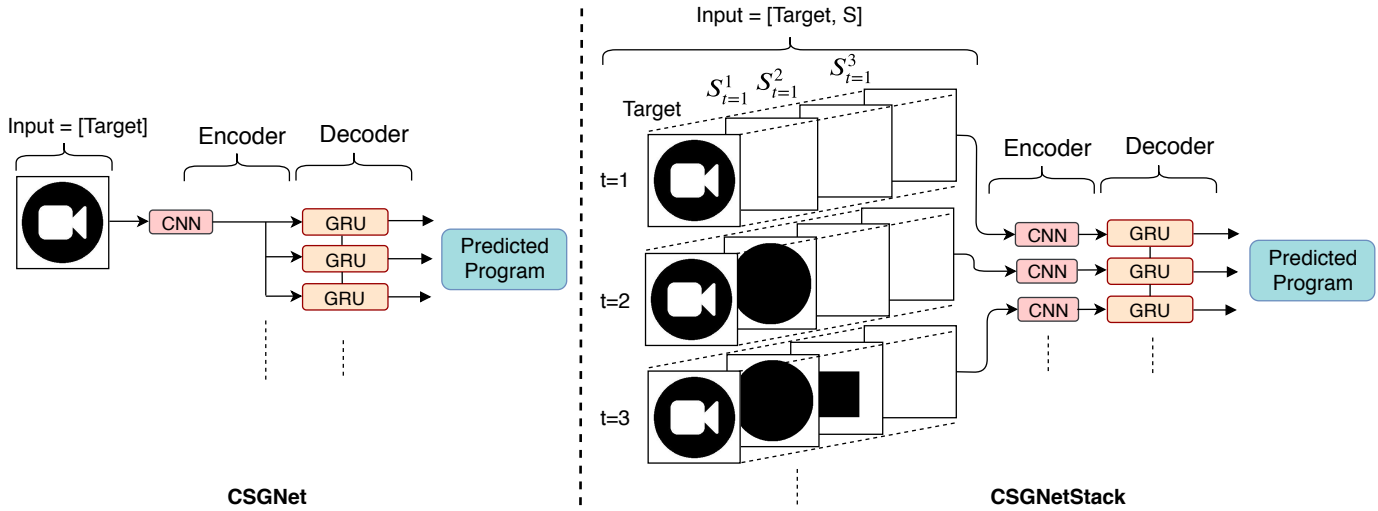


Fig. 3: **Two proposed architectures of our neural shape parser CSGNET (left), CSGNETSTACK (right)**. CSGNet takes the target shape as input and encodes it using a CNN, whereas in CSGNETSTACK, the target shape is concatenated with stack S_t along the channel dimension and passes as input to the CNN encoder at every time step. Empty entries in the stack are shown in white.

of shapes is concatenated with the target shape, all stored as binary maps, along the channel dimension. The concatenated map is processed by the network at the next time step. Instead of taking all elements of the stack, which vary in number depending on the generated program, we only take the top-K maps of the stack. Empty entries in the stack are represented as all-zero maps (see also Figure 3, right). At the first time step, the stack is empty, so all K maps are zero. While the stack contains complete information about the program execution at any point in time, it can grow arbitrarily deep. Keeping the top-K elements of the stack provides a way to trade-off the computational and memory requirements with the amount of information about the program execution.

In our implementation, the parser π takes $Z = [I, S]$ as input of size $64\times64\times(K+1)$ for $2D$ networks and $64\times64\times64\times(K+1)$ for $3D$ networks, where I is the input shape, S is the execution stack of the renderer, and K is the size of the stack. The number of channels is $(K + 1)$ since the target shape, also represented as 64^2 (or 64^3 in 3D), is concatenated with the stack. Details of the architecture are described in Section 4. Similarly to the basic CSGNET architecture, the encoder takes Z as input and yields a fixed length encoding $\Phi(Z)$, which is passed as input to the decoder Θ to produce a probability distribution over programs P. The stack-based network can be written as $\pi(Z) = \Theta \circ \Phi(Z)$. We call this stack based architecture CSGNETSTACK. The difference between the two architectures is illustrated in Figure 3.

Grammar. The space of programs can be efficiently described according to a context-free grammar [60]. A context-free grammar is a formal grammar when its production rules can be applied regardless of the context of its non-terminal symbols. For example, in constructive solid geometry the instructions consist of drawing primitives (eg, spheres, cubes, cylinders, etc) and performing boolean operations described as a grammar with the following production rules:

$$
\begin{aligned}
S &\rightarrow E \\
E &\rightarrow E\ E\ T \mid P \\
T &\rightarrow \texttt{OP}_1 | \texttt{OP}_2 | \ldots | \texttt{OP}_m \\
P &\rightarrow \texttt{SHAPE}_1 | \texttt{SHAPE}_2 | \ldots | \texttt{SHAPE}_n
\end{aligned}
$$

Each rule indicates possible derivations of a non-terminal symbol separated by the $|$ symbol. Here S is the start symbol, $\texttt{OP}_i$ is chosen from a set of defined modeling operations and the $\texttt{SHAPE}_i$ is a primitive chosen from a set of basic shapes at different positions, scales, orientations, etc. Instructions can be written in a standard post-fix notation, *e.g.*, $\texttt{SHAPE}_1\texttt{SHAPE}_2\texttt{OP}_1\texttt{SHAPE}_3\texttt{OP}_2$, which can be written in in-fix notation as: $(\texttt{SHAPE}_1\ \texttt{OP}_1\ \texttt{SHAPE}_2)\ \texttt{OP}_2\ \texttt{SHAPE}_3$. Table 4 shows an example of a program predicted by the network and corresponding rendering process.

3.1 Learning

Given the input shape I and execution stack S of the renderer, the parser network π generates a program that minimizes a reconstruction error between the shape produced by executing the program and a target shape. Note that not all programs are valid. Our learning incorporates rewards promoting the generation of programs that are both valid and capture the target shape well.

3.1.1 Supervised learning

When target programs are available both CSGNET and CSGNETSTACK variants can be trained with standard supervised learning techniques. Training data consists of N shapes, P corresponding programs, and also in the case of CSGNETSTACK S stacks, program triplets (I^i, S^i, P^i), $i = 1, \ldots, N$. The ground-truth program P^i can be written as a sequence of instructions g^i_1, g^i_2 .. $g^i_{T_i}$, where T_i is the length of the program P^i. Similarly, in the case of CSGNETSTACK, the S^i can be written as sequence of states of stack s^i_1, s^i_2 .. $s^i_{T_i}$ used by the rendering engine while executing the instructions in program P^i. Note that while training in supervised setting, the stack s_t is generated by the renderer while executing ground truth instructions $g_{1:t}$, but during inference time, the stack is generated by the renderer while executing the predicted instructions. For both network variants, the RNN produces a categorical distribution π for both variants.

The parameters θ for either variant can be learned to maximize the log-likelihood of the ground truth instructions:

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \sum_{t=1}^{T_i} \log \pi_\theta(g^i_t | g^i_{1:t-1}, s^i_{1:t-1}, I^i) \qquad (1)$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPAMI.2020.3044749, IEEE Transactions on Pattern Analysis and Machine Intelligence

Instruction	Execution	Stack
`circle(32,32,28)`	`push circle(32,32,28)`	`[P1]`
`square(32,40,24)`	`push square(32,40,24)`	`[P2 P1]`
`circle(48,32,12)`	`push circle(48,32,12)`	`[P3 P2 P1]`
`circle(24,32,16)`	`push circle(24,32,16)`	`[P4 P3 P2 P1]`
`union`	`A=pop; B=pop; push(B∪A)`	`[E1 P2 P1] // E1=P3∪P4`
`intersect`	`A=pop; B=pop; push(B∩A)`	`[E2 P1] // E2=P2∩E1`
`subtract`	`A=pop; B=pop; push(B-A)`	`[Out] // Out=P1-E2`

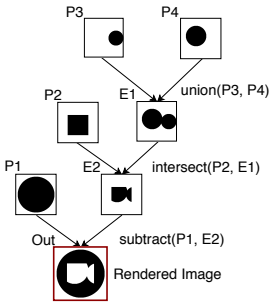


Fig. 4: **Example program execution.** Each row in the table from the top shows the instructions, program execution, and the current state of the stack of the shift-reduce CSG parser. On the right is a graphical representation of the program. An instruction corresponding to a primitive leads to `push` operation on the stack, while an operator instruction results in popping the top two elements of the stack and pushing the result of applying this operator.

3.1.2 Learning with policy gradients

Without target programs one can minimize a reconstruction error between the shape obtained by executing the program and the target. However, directly minimizing this error using gradient-based techniques is not possible since the output space is discrete and execution engines are typically not differentiable. Policy gradient techniques [4] from the reinforcement learning (RL) literature can instead be used in this case.

Concretely, the parser π_θ, that represents a policy network, can be used to sample a program $y = (a_1, a_2 \,..\, a_T)$ conditioned on the input shape I, and in the case of CSGNetStack, also on the stack $S = (s_1, s_2 \,..\, s_T)$. Note that while training using policy gradient and during inference time, the stack s_t is generated by the renderer while executing predicted instructions by the parser since ground-truth programs are unavailable. Then a reward R can be estimated by measuring the similarity between the generated image $\hat{I}$ obtained by executing the program and the target shape I. With this setup, we want to learn the network parameters θ that maximize the expected rewards over programs sampled under the predicted distribution $\pi_\theta(y|S, I)$ across images I sampled from a distribution $\mathcal{D}$:

$$\mathbb{E}_{I\sim\mathcal{D}}\left[J_\theta(I)\right] = \mathbb{E}_{I\sim\mathcal{D}} \sum_{t=1}^{T} \mathbb{E}_{y_t\sim\pi_\theta(y|s_{1:t-1},I)}\left[R\right]$$

The outer expectation can be replaced by a sample estimate on the training data. The gradient of the inner expectation can be obtained by rearranging the equation as [1]:

$$\nabla_\theta J_\theta(I) = \nabla_\theta \sum_y \pi_\theta(y) R = \sum_y \nabla_\theta \log \pi_\theta(y) \left[\pi_\theta(y) R\right]$$

Here we use the identity $\nabla_\theta \pi_\theta(y) = \pi_\theta(y) \nabla_\theta \log \pi_\theta(y)$. It is often intractable to compute the expectation $J_\theta(I)$ since the space of programs is very large. Hence, the expectation must be approximated. The REINFORCE algorithm computes a Monte-Carlo estimate (see also [4], [61] for derivations and explanation of the policy gradient algorithm). This is expressed as:

$$\nabla_\theta J_\theta(I) = \frac{1}{M} \sum_{m=1}^{M} \sum_{t=1}^{T} \nabla \log \pi_\theta(\hat{a}_t^m | \hat{a}_{1:t-1}^m, \hat{s}_{1:t-1}^m, I) R^m$$

by sampling M programs from the policy π_θ. Each program y^m is obtained by sampling instructions $\hat{a}_{t=1:T}^m$ from the distribution $\hat{a}_t^m \sim \pi_\theta(a_t | \hat{a}_{1:t-1}^m; \hat{s}_{1:t-1}^m, I)$ at every time step t until the stop symbol (`EOS`) is sampled. The reward R^m is calculated by executing the program y^m. Sampling-based estimates typically have high variance that can be reduced by subtracting a baseline without changing the bias as:

$$\nabla_\theta J_\theta(I) = \frac{1}{M} \sum_{m=1}^{M} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\hat{a}_t^m | \hat{a}_{1:t-1}^m, \hat{s}_{1:t-1}^m, I)(R^m - b) \tag{2}$$

A good choice of the baseline is the expected value of returns starting from t [4], [62]. We compute the baseline as the running average of past rewards.

Reward. The rewards should be primarily designed to encourage visual similarity of the generated program with the target. Visual similarity between two shapes is measured using the Chamfer distance (CD) between points on the silhouettes of each shape. We focus on the silhouettes because these tend to be more related to the perceptual similarity of shapes [63]. The CD is between two point sets, $\mathbf{x}$ and $\mathbf{y}$, is defined as follows:

$$Ch(\mathbf{x}, \mathbf{y}) = \frac{1}{2|\mathbf{x}|} \sum_{x\in\mathbf{x}} \min_{y\in\mathbf{y}} \|x - y\|_2 + \frac{1}{2|\mathbf{y}|} \sum_{y\in\mathbf{y}} \min_{x\in\mathbf{x}} \|x - y\|_2$$

The points are scaled by the image diagonal, thus $Ch(\mathbf{x}, \mathbf{y}) \in [0, 1] \;\; \forall \mathbf{x}, \mathbf{y}$. The distance can be efficiently computed using distance transforms. In our implementation, we also set a maximum length T for the induced programs to avoid having too long or redundant programs (*e.g.*, repeating the same modeling instructions over and over again). We then define the reward as:

$$R = \begin{cases} f\left(Ch(\texttt{Edge}(I), \texttt{Edge}(\Re(y))\right), & y \text{ is valid} \\ 0, & y \text{ is invalid} \end{cases}$$

where f is a reward shaping function and $\Re$ is the CSG rendering engine that renders the program y into a binary image. Note that a valid program follows the grammar described in the Section 4.1, which can be verified by the execution engine. Since invalid programs get zero reward, the maximum length constraint on the programs helps the network to produce shorter programs with high rewards. We use maximum length $T = 13$ in all of our RL experiments. The function f shapes the CD as $f(x) = (1 - x)^\gamma$ with an exponent $\gamma > 0$. Higher values of γ makes the reward closer to zero, thereby making the network to produce programs with smaller CD. Table 1 (left) shows the dynamics of reward shaping function with different γ value and (right) shows that

1. conditioning on stack and input image is removed for the sake of brevity.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPAMI.2020.3044749, IEEE Transactions on Pattern Analysis and Machine Intelligence

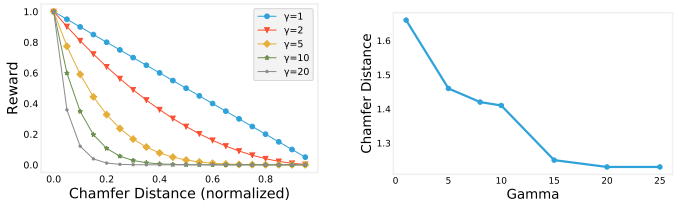


TABLE 1: **Reward shaping.** (Left) We visualize the skewness introduced by the γ in the reward function. (Right) Larger γ value produces smaller CD (in number of pixels) when our model is trained using REINFORCE.

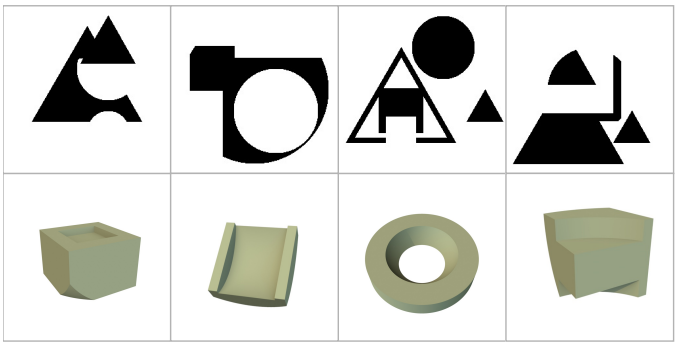


Fig. 5: **Samples of our synthetically generated programs.** 2D samples are in the top row and 3D samples in the bottom. For clarity, the shapes are rendered in their original, high-resolution mesh format before voxelization.

increasing γ values decreases the average CD calculated over the test set. We choose $\gamma = 20$ in our experiments, as this gives best performance on our validation set as shown in Table 1.

3.2 Inference

Greedy decoding and beam search. Estimating the most likely program given an input is intractable using RNNs. Instead one usually employs a greedy decoder that picks the most likely instruction at each time step. An alternate is to use a beam search procedure that maintains the k-best likely sequences at each time step. In our experiments we report results with varying beam sizes.

Visually-guided refinement. Both parser variants produce a program with a discrete set of primitives. However, further refinement can be done by directly optimizing the position and size of the primitives to maximize the reward. The refinement step keeps the program structure of the program and primitive type fixed but uses a heuristic algorithm [64] to optimize the parameters using feedback from the rendering engine. In our experiments, we observed that the algorithm converges to a local minima in about 10 iterations of refinement and consistently improves the results.

4 Experiments

We describe our experiments on different datasets exploring the generalization capabilities of our network variants (CSGNet and CSGNetStack). We first describe our datasets: (i) an automatically generated dataset of 2D and 3D shapes based on synthetic generation of CSG programs, (ii) 2D CAD shapes mined from the web where ground-truth programs are not available, and (iii) logo images mined also from the web where ground-truth programs are also not available. Below we discuss our qualitative and quantitative results on the above dataset.

Program Length	2D			3D		
	Train	Val	Test	Train	Val	Test
3	25k	5k	5k	100k	10k	20k
5	100k	10k	50k	200k	20k	40k
7	150k	20k	50k	400k	40k	80k
9	250k	20k	50k	-	-	-
11	350k	20k	100k	-	-	-
13	350k	20k	100k	-	-	-

TABLE 2: **Statistics of our 2D and 3D synthetic dataset.**

4.1 Datasets

To train our network in the supervised learning setting, we automatically created a large set of 2D and 3D CSG-based synthetic programs according to the grammars described below.

Synthetic 2D shapes. We sampled derivations of the following CSG grammar to create our synthetic dataset in the 2D case:

$$
\begin{aligned}
&S \rightarrow E; \\
&E \rightarrow EET \mid P(L, R); \\
&T \rightarrow intersect \mid union \mid subtract; \\
&P \rightarrow square \mid circle \mid triangle; \\
&L \rightarrow \Big[8:8:56\Big]^2; \;\; R \rightarrow \Big[8:4:32\Big].
\end{aligned}
$$

Primitives are specified by their type: *square*, *circle*, or *triangle*, locations L and circumscribing circle of radius R on a canvas of size 64×64. There are three boolean operations: $intersect$, $union$, and $subtract$. L is discretized to lie on a square grid with spacing of 8 units and R is discretized with spacing of 4 units. The *triangles* are assumed to be upright and equilateral. The synthetic dataset is created by sampling random programs containing different number of primitives from the above grammar, constraining the distribution of various primitive types and operation types to be uniform. We also ensure that no duplicate programs exist in our dataset. The primitives are rendered as binary images and the programs are executed on a canvas of 64×64 pixels. Samples from our dataset are shown in Figure 5. Table 2 provides details about the size and splits of our dataset.

Synthetic 3D shapes. We sampled derivations of the following grammar in the case of 3D CSG:

$$
\begin{aligned}
&S \rightarrow E; \;\; E \rightarrow EET; \\
&E \rightarrow sp(L, R) \mid cu(L, R) \mid cy(L, R, H) \\
&T \rightarrow intersect \mid union \mid subtract; \\
&L \rightarrow \Big[8:8:56\Big]^3 \\
&R \rightarrow \Big[8:4:32\Big]; \;\; H \rightarrow \Big[8:4:32\Big].
\end{aligned}
$$

The operations are same as in the 2D case. Three basic solids are denoted by 'sp': Sphere, 'cu': Cube, 'cy': Cylinder. L represents the center of primitive in a 3D voxel grid. R specifies radius of sphere and cylinder, or the size of cube. H is the height of cylinder. The primitives are rendered as voxels and the programs are executed on a 3D volumetric grid of size 64×64

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPAMI.2020.3044749, IEEE Transactions on Pattern Analysis and Machine Intelligence

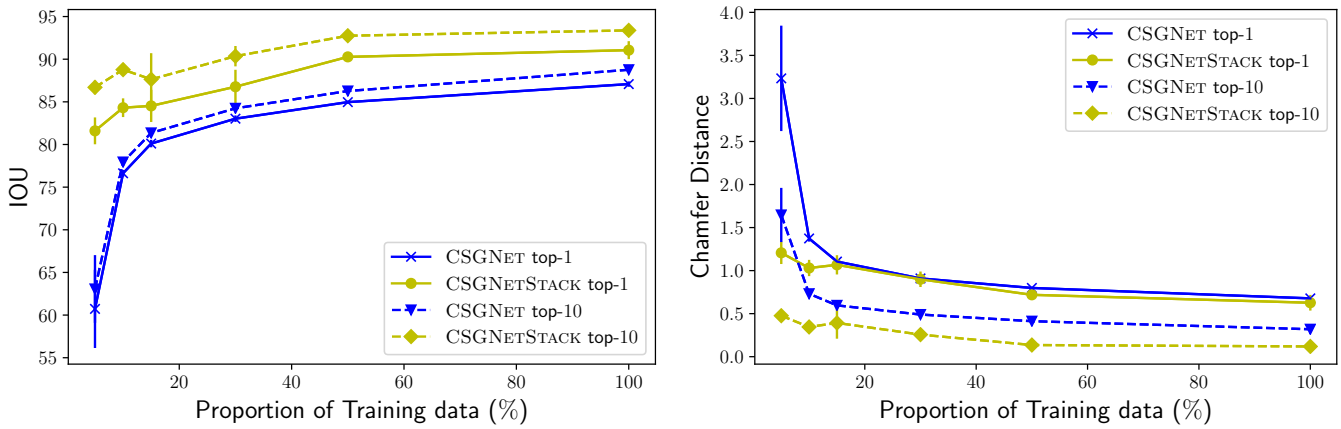


Fig. 6: **Performance (Left: IOU, Right: chamfer distance) of models by changing training size on our synthetic dataset.** Training is done using $x\%$ of the complete dataset, where x is shown on the horizontal axis. The top-k beam sizes used during decoding at test time are shown in the legend. The performance of CSGNET (our basic non-stack neural shape parser) is shown in blue and the performance of CSGNETSTACK (our variant that uses the execution stack) is shown in lime. The above plots show the average of the metrics evaluated at 4 different training runs.

$\times$ 64. We used the same random sampling method as used for the synthetic 2D dataset, resulting in 3D CSG programs. 3D shape samples are shown in Figure 5.

2D CAD shapes. We collected $8K$ CAD shapes from the Trimble 3DWarehouse dataset [65] in three categories: chair, desk and lamps. We rendered the CAD shapes into 64×64 binary masks from their front and side views. In Section 4, we show that the rendered shapes can be parsed effectively through our visual program induction method. We split this dataset into $5K$ shapes for training, $1.5K$ validation and $1.5K$ for testing.

Web logos. We mined 20 binary logos from the web that can be modeled using the primitives in our output shapes. We test our approach on these logos without further training or fine-tuning our net on this data.

4.2 Implementation details

2D shape parsing. Our encoder is based on an image-based convnet in the case of 2D inputs. In the case of CSGNETSTACK, the input to the network is a fixed size stack along with target image concatenated along the channel dimension, resulting in an the input tensor of size $64 \times 64 \times (K+1)$, where K is the number of used maps in the stack (stack size). In the architecture without stack (CSGNET), K is simply set to 0. The output of the encoder is passed as input to our GRU-based decoder at every program step. The hidden state of our GRU units is passed through two fully-connected layers, which are then converted into a probability distribution over program instructions through a classification layer. For the 2D CSG there are 400 unique instructions corresponding to 396 different primitive types, discrete locations and sizes, the 3 boolean operations and the stop symbol.

3D shape parsing. In the case of 3D shapes, the encoder is based on an volumetric, voxel-based convnet. 3D-CSGNETSTACK concatenates the stack with the target shape along the channel dimension, resulting in an input tensor of size $64 \times 64 \times 64 \times (K+1)$,

Method	IOU (k=1) ↑	IOU (k=10) ↑	CD (k=1) ↓	CD (k=10) ↓
NN	73.9	-	1.93	-
CSGNET	86.77	88.74	0.70	0.32
CSGNETSTACK	**91.33**	**93.45**	**0.60**	**0.12**

TABLE 3: **Comparison of a NN baseline with the supervised network without stack (CSGNET) and with stack (CSGNETSTACK) on the synthetic 2D dataset.** Results are shown using Chamfer Distance (CD) and IOU metric by varying beam sizes (k) during decoding. CD is in number of pixels.

where K is the number of used maps in the stack (stack size). In the architecture without stack (3D-CSGNET), K is simply set to 0. The encoder comprises of multiple layers of $3D$ convolutions yielding a fixed size encoding vector. Similarly to the $2D$ case, the GRU-based decoder takes the output of the encoder and sequentially produces the program instructions. In this case, there are 6635 unique instructions with 6631 different types of primitives with different sizes and locations, plus 3 boolean modeling operations and a stop symbol.

During training, on synthetic dataset, we sample images/3D shapes rendered from programs of variable length (up to 13 for 2D and up to 7 for 3D dataset) from training dataset from Table 2. More details about the architecture of our encoder and decoder (number and type of layers) are provided in the supplementary material.

For supervised learning, we use the Adam optimizer [66] with learning rate 0.001 and dropout of 0.2 in non-recurrent network connections. For reinforcement learning, we use stochastic gradient descent with 0.9 momentum, 0.01 learning rate, and with the same dropout as above.

4.3 Results

We evaluate our network variants in two different ways: (i) as models for inferring the entire program, and (ii) as models for inferring primitives, *i.e.*, as object detectors.

4.3.1 *Inferring programs*

Evaluation on the synthetic 2D shapes. We perform supervised learning to train our stack-based network CSGNETSTACK and

Method	Train	Test	CD (@refinement iterations) ↓					
			i=0	i=1	i=2	i=4	i=10	i=∞
NN	-	-	1.92	1.22	1.13	1.08	1.07	1.07
CSGNET	Supervised	k=1	2.45	1.2	1.03	0.97	0.96	0.96
CSGNET	Supervised	k=10	1.68	0.79	0.67	0.63	0.62	0.62
CSGNETSTACK	Supervised	k=1	3.98	2.66	2.41	2.29	2.25	2.25
CSGNETSTACK	Supervised	k=10	1.38	0.56	0.45	0.40	0.39	0.39
CSGNET	RL	k=1	1.40	0.71	0.63	0.60	0.60	0.60
CSGNET	RL	k=10	1.19	0.53	0.47	0.41	0.41	0.41
CSGNETSTACK	RL	k=1	1.27	0.67	0.60	0.58	0.57	0.57
CSGNETSTACK	RL	k=10	**1.02**	**0.48**	**0.43**	**0.35**	**0.34**	**0.34**

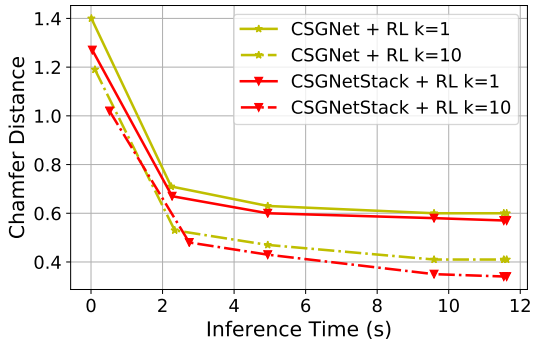


TABLE 4: **Comparison of various approaches on the CAD shape dataset**. CSGNET: neural shape parser without stack, CSGNETSTACK: parser with stack, NN: nearest neighbor. Left: Results are shown with different beam sizes (k) during decoding. Fine-tuning using RL improves the performance of both network, with CSGNETSTACK perfoming the best. Increasing the number of iterations (i) of visually guided refinement during testing also improves results significantly. $i = \infty$ corresponds to running visually guided refinement till convergence. Right: Inference time for different methods. Increasing number of iterations of visually guided refinement improves the performance, with least CD in a given inference time is produced by Stack based architecture. CD metric is in number of pixels.

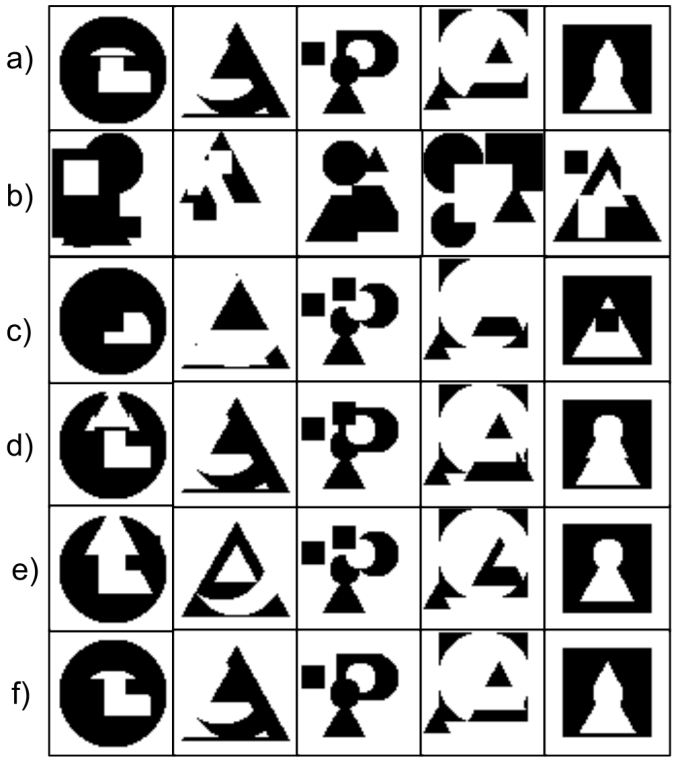


Fig. 7: **Comparison of performance on synthetic 2D dataset.** a) Input image, b) NN-retrieved image, c) top-1 prediction of CSGNET, d) top-1 prediction of CSGNETSTACK, e) top-10 prediction of CSGNET and f) top-10 prediction of CSGNETSTACK.

the non-stack-based network CSGNET on the training split of this synthetic dataset, and evaluate performance on its test split under different beam sizes. We compare with a baseline that retrieves a program in the training split using a Nearest Neighbor (NN) approach. In NN setting, the program for a test image is retrieved by taking the program of the train image that is most similar to the test image using the IOU metric.

Table 3 compares CSGNETSTACK, CSGNET, and a NN baseline using the Chamfer distance and IOU between the test target and predicted shapes using the complete synthetic dataset. Our parser is able to outperform the NN method. One would expect that NN would perform well here because the size of the training set is large. However, our results indicate that our compositional parser is better at capturing shape variability, which is still significant in this dataset. Results are also shown with increasing beam sizes (k) during decoding, which consistently improves performance. Figure 7 also shows the programs retrieved through NN and our generated program for a number of characteristic examples in our test split of our synthetic dataset.

We also examine the learning capability of CSGNETSTACK with significantly less synthetic training dataset in comparison to CSGNET in the Figure 6. With just 5% of the total dataset, CSGNETSTACK performs 80% IOU (1.3 CD) in comparison to 70% IOU (1.7 CD) using CSGNET. The CSGNETSTACK continues to perform better compared to CSGNET in the case of more training data. This shows that incorporating the extra knowledge in the form of an execution stack based on the proposed architecture makes it easier to learn to parse shapes.

Evaluation on 2D CAD shapes. For this dataset, we report results on its test split under two conditions: (i) when training our network only on synthetic data, and (ii) when training our network on synthetic data and also fine-tuning it on the training split of rendered CAD dataset using policy gradients.

Table 4 shows quantitative results on this dataset. We first compare with the NN baseline. For any shape in this dataset, where ground truth program is not available, NN retrieves a shape from synthetic dataset and we use the ground truth program of the retrieved synthetic shape for comparison.

We then list the performance of CSGNETSTACK and CSGNET trained in a supervised manner only on our synthetic dataset. Further training with Reinforcement Learning (RL) on the training split of the 2D CAD dataset improves the results significantly and outperforms the NN approach by a considerable margin. This also shows the advantage of using RL, which trains the shape parser without ground-truth programs. The stack based network CSGNETSTACK performs better than CSGNET showing

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPAMI.2020.3044749, IEEE Transactions on Pattern Analysis and Machine Intelligence

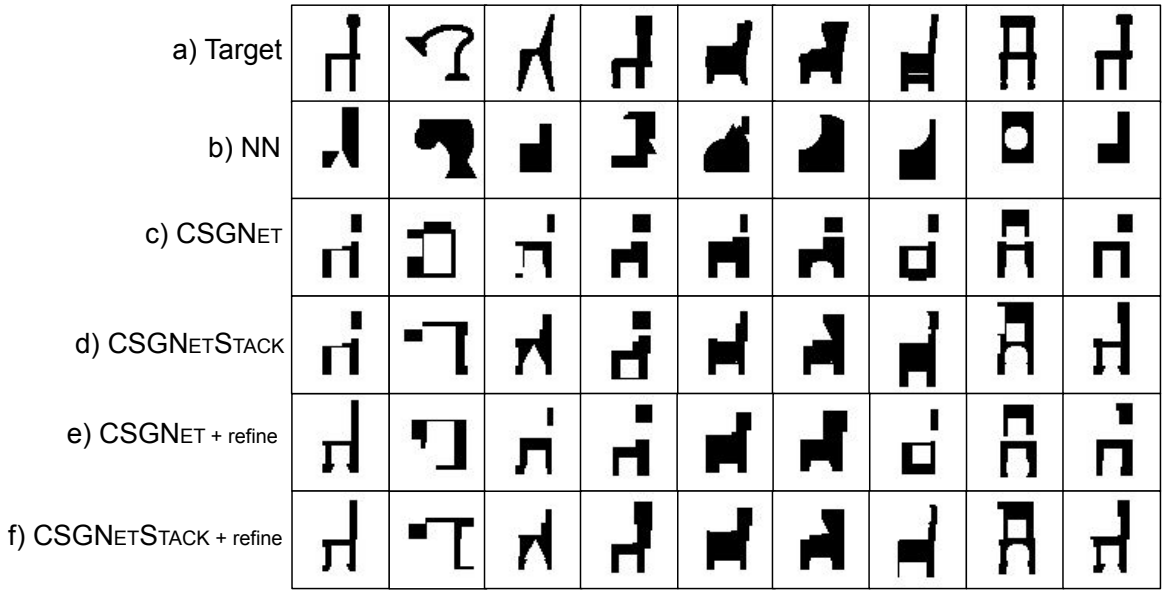


Fig. 8: **Comparison of performance on the 2D CAD dataset**. a) Target image, b) NN retrieved image, c) best result from beam search on top of CSGNET fine-tuned with RL, d) best result from beam search on top of CSGNETSTACK fine-tuned with RL, and refining results using the visually guided search on the best beam result of CSGNET (e) and CSGNETSTACK (f).

Method	NN	3D-CSGNET			3D-CSGNETSTACK		
		k=1	k=5	k=10	k=1	k=5	k=10
IOU (%) ↑	73.2	80.1	85.3	89.2	81.5	86.9	90.5
CD ↓	2.53	1.86	1.19	0.93	1.71	1.01	0.81

TABLE 5: **Comparison of the supervised network (3D-CSGNETSTACK and 3D-CSGNET) with NN baseline on the 3D dataset**. Results are shown using IOU(%) and Chamfer distance (CD) metrics, and varying beam sizes (k) during decoding. CD has been multiplied by 100.

better generalization on the new dataset. We note that directly training the network using RL alone does not yield good results which suggests that the two-stage learning (supervised learning and RL) is important. Finally, optimizing the best beam search program with visually guided refinement yielded results with the smallest Chamfer Distance. Figure 8 shows a comparison of the rendered programs for various examples in the test split of the 2D CAD dataset for variants of our network. Visually guided refinement on top of beam search of our two stage-learned network qualitatively produces results that best match the input image.

We also show an ablation study indicating how much pretraining on the synthetic dataset is required to perform well on the CAD dataset in Figure 9. With just 5% of the synthetic dataset based pretraining, CSGNETSTACK gives 60% IOU (and 1.3 CD) in comparison to 46% IOU (and 1.9 CD), which shows the faster learning capability of our stack based architecture. Increasing the synthetic training size used in pretraining shows slight decrease in performance for the CSGNETSTACK network after 15%, which hints at the overfitting of the network on the synthetic dataset domain.

Evaluation on Logos. We experiment with the logo dataset described in Section 4.1 (none of these logos participate in training). Outputs of the induced programs parsing the input logos are shown in Figure 10. In general, our method is able to parse logos into primitives well, yet performance can degrade when long programs are required to generate them, or when they contain shapes that are very different from our used primitives.

Evaluation on Synthetic 3D CSG. Finally, we show that our approach can be extended to 3D shapes. In the 3D CSG setting we use 3D-CSG dataset as described in the Section 4.1. We train a stack based 3D-CSGNETSTACK network that takes $64 \times 64 \times 64 \times (K+1)$ voxel representation of input shape concatenated with voxel representation of stack. The input to our 3D-CSGNET are voxelized shapes in a $64 \times 64 \times 64$ grid. Our output is a 3D CSG program, which can be rendered as a high-resolution polygon mesh (we emphasize that our output is not voxels, but CSG primitives and operations that can be computed and rendered accurately). Figure 11 show pairs of input voxel grids and our output shapes from the test split of the 3D dataset. The quantitative results are shown in the Table 5, where we compare our 3D-CSGNETSTACK and 3D-CSGNET networks at different beam search decodings with the NN method, using both the IOU and Chamfer distance metrics. Chamfer distance is computed using Eq. 3 by sampling 5k points on ground truth and predicted surface. The stack-based network also improves the performance over the non-stack variant. The results indicate that our method is promising in inducing correct programs for 3D shapes, which also has the advantage of accurately reconstructing the voxelized surfaces into high-resolution surfaces.

4.3.2 *Primitive detection*

Successful program induction for a shape requires not only predicting correct primitives but also correct sequences of operations to combine these primitives. Here we evaluate the shape parser as a primitive detector (*i.e.*, we evaluate the output primitives of our program, not the operations themselves). This allows us to directly compare our approach with bottom-up object detection techniques.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPAMI.2020.3044749, IEEE Transactions on Pattern Analysis and Machine Intelligence

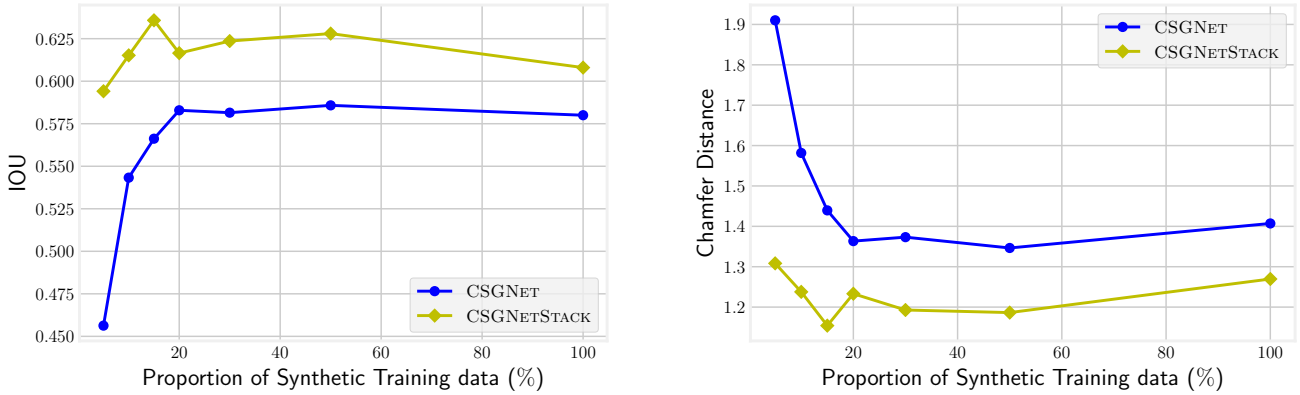


Fig. 9: **Performance (Left: IOU, Right: chamfer distance) of CSGNET and CSGNETSTACK on the test split of the 2D CAD dataset wrt the size of the synthetic dataset used to pre-train the two architectures.** Pre-training is done using $x\%$ of the complete synthetic dataset (x is shown on the horizontal axis) and fine-tuning is done on the complete CAD dataset. CSGNETSTACK performs better while using less proportion of the synthetic dataset for pretraining. Increasing the size of pretraining dataset beyond 15% leads to decrease in performance, which hints at slight overfitting on the synthetic dataset domain.

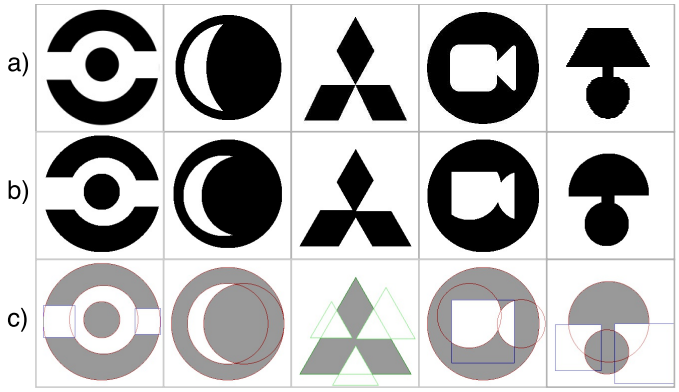


Fig. 10: **Results for our logo dataset**. a) Target logos, b) output shapes from CSGNET and c) inferred primitives from output program. Circle primitives are shown with red outlines, triangles with green and squares with blue.

Method	Circle	Square	Triangle	Mean	Speed (im/s)
Faster R-CNN	87.4	71.0	81.8	80.1	5
CSGNET, $k = 10$	86.7	79.3	83.1	83.0	80
CSGNET, $k = 40$	**88.1**	**80.7**	**84.1**	**84.3**	20

TABLE 6: **MAP of detectors on the synthetic 2D shape dataset.** We also report detection speed measured as images/second on a NVIDIA 1070 GPU.

In particular we compare against Faster R-CNNs [7], a state-of-the-art object detector. The Faster R-CNN is based on the VGG-M network [67] and is trained using bounding-box and primitive annotations based on our 2D synthetic training dataset. At test time the detector produces a set of bounding boxes with associated class scores. The models are trained and evaluated on 640×640 pixel images. We also experimented with bottom-up approaches for primitive detection based on Hough transform [68] and other rule-based approaches. However, our experiments indicated that the Faster R-CNN was considerably better.

For a fair comparison, we obtain primitive detections from CSGNET trained on the 2D synthetic dataset only (same as the Faster R-CNN). To obtain detection scores, we sample k programs with beam-search decoding. The primitive score is the fraction of times it appears across all beam programs. This is a Monte Carlo estimate of our detection score. The accuracy can be measured through standard evaluation protocols for object detection (similar to those in the PASCAL VOC benchmark). We report the Mean Average Precision (MAP) for each primitive type using an overlap threshold between the predicted and the true bounding box of 0.5 intersection-over-union. Table 6 compares the parser network to the Faster R-CNN approach.

Our parser clearly outperforms the Faster R-CNN detector on the squares and triangles category. With larger beam search, we also produce slighly better results for circle detection. Interestingly, our parser is considerably faster than Faster R-CNN tested on the same GPU.

5 LIMITATIONS AND CONCLUSION

We believe that our work represents a step towards neural generation of modeling programs given target visual content, which we believe is quite ambitious and hard problem. We demonstrated that the model generalizes across domains, including logos, 2D silhouettes, and 3D CAD shapes. It also is an effective primitive detector in the context of 2D shape primitive detection.

One might argue that the 2D images and 3D shapes considered in this work are relatively simple in structure or geometry. However, we would like to point out that even in this ostensibly simple application scenario (i) our method demonstrates competitive or even better results than state-of-the-art object detectors, and most importantly (ii) the problem of generating programs using neural networks was far from trivial to solve: based on our experiments, a combination of memory-enabled networks, supervised and RL strategies, along with beam and local exploration of the state space all seemed necessary to produce good results.

As future work, we would like to generalize our approach to longer programs with much larger spaces of parameters in the modeling operations and more sophisticated reward functions balancing perceptual similarity to the input image and program length. Our method is currently limited in its capability to generate 3D shapes, since the supported resolution is low due to

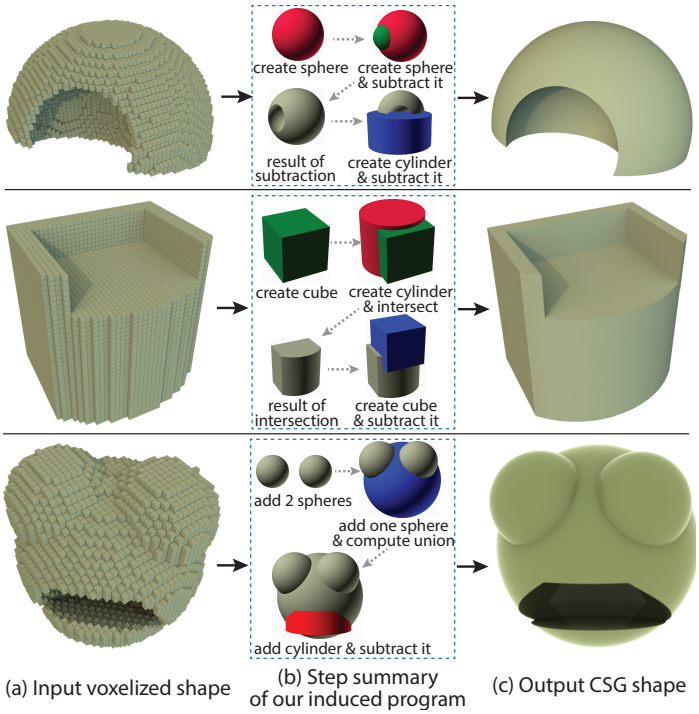


Fig. 11: **Qualitative performance of 3D-CSGNET.** a) Input voxelized shape, b) Summarization of the steps of the program induced by 3D-CSGNET in the form of intermediate shapes, c) Final output created by executing induced program.

the voxel representation we use in our encoder. Sparser shape representations [69] could help extending our network to handle more challenging 3D cases and datasets, such as ShapeNet [70] and ABC [71]. Another limitation is that our current control of the CSG program size is crude; it is based only on an upper bound of program size and a zero reward for invalid programs, which often occur with larger number of program instructions. Investigating more sophisticated complexity penalties could help promoting right-sized programs. Other promising direction is alternate strategies for combining bottom-up proposals and top-down approaches for parsing shapes, in particular, approaches based on constraint satisfaction and generic optimization.

Acknowledgments. The project is supported in part by grants from the National Science Foundation (NSF) CHS-1422441, CHS-1617333, IIS-1617917. We also acknowledge the MassTech collaborative grant for funding the UMass GPU cluster.

REFERENCES

[1] I. Biederman, "Recognition-by-Components: A Theory of Human Image Understanding," *Psychological Review*, vol. 94, no. 2, 1987.
[2] D. H. Laidlaw, W. B. Trumbore, and J. F. Hughes, "Constructive solid geometry for polyhedral objects," in *Proc. SIGGRAPH*, 1986.
[3] S. Gershman and N. D. Goodman, "Amortized inference in probabilistic reasoning," in *Proceedings of the Thirty-Sixth Annual Conference of the Cognitive Science Society*, 2014.
[4] R. J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," *Machine Learning*, vol. 8, no. 3-4, pp. 229–256, 1992.
[5] E. W. Dijkstra, "Recursive programming," *Numer. Math.*, vol. 2, no. 1, 1960.
[6] A. Y. Ng, D. Harada, and S. J. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *Proc. ICML*, 1999.
[7] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," in *Proc. NIPS*, 2015.
[8] G. Sharma, R. Goyal, D. Liu, E. Kalogerakis, and S. Maji, "Csgnet: Neural shape parser for constructive solid geometry," *CoRR*, vol. abs/1712.08290, 2017.
[9] "Pytorch," https://pytorch.org.
[10] M. A. Fischler and R. A. Elschlager, "The representation and matching of pictorial structures," *IEEE Transactions on computers*, vol. 100, no. 1, pp. 67–92, 1973.
[11] P. F. Felzenszwalb and D. P. Huttenlocher, "Pictorial structures for object recognition," *IJCV*, vol. 61, no. 1, pp. 55–79, 2005.
[12] Y. Yang and D. Ramanan, "Articulated pose estimation with flexible mixtures-of-parts," in *Proc. CVPR*, 2011.
[13] L. Bourdev, S. Maji, T. Brox, and J. Malik, "Detecting people using mutually consistent poselet activations," in *Proc. ECCV*, 2010.
[14] V. Shapiro and D. L. Vossler, "Construction and optimization of csg representations," *Comput. Aided Des.*, vol. 23, no. 1, 1991.
[15] S. F. Buchele and R. H. Crawford, "Three-dimensional halfspace constructive solid geometry tree construction from implicit boundary representations," in *Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications*, 2003.
[16] V. Shapiro and D. L. Vossler, "Separation for boundary to csg conversion," *ACM Trans. Graph.*, vol. 12, no. 1, 1993.
[17] K. Hamza and K. Saitou, "Optimization of constructive solid geometry via a tree-based multi-objective genetic algorithm," in *Genetic and Evolutionary Computation*, 2004.
[18] D. Weiss, "Geometry-based structural optimization on cad specification trees, phd dissertation, eth zurich," 2009.
[19] P.-A. Fayolle and A. Pasko, "An evolutionary approach to the extraction of object construction trees from 3d point clouds," *Computer-Aided Design*, vol. 74, pp. 1 – 17, 2016.
[20] T. Du, J. P. Inala, Y. Pu, A. Spielberg, A. Schulz, D. Rus, A. Solar-Lezama, and W. Matusik, "Inversecsg: Automatic conversion of 3d models to csg trees," *ACM Trans. Graph.*, vol. 37, no. 6, Dec. 2018.
[21] C. A. Vanegas, I. Garcia-Dorado, D. G. Aliaga, B. Benes, and P. Waddell, "Inverse Design of Urban Procedural Models," *ACM Transactions on Graphics*, vol. 31, no. 6, 2012.
[22] O. Stava, S. Pirk, J. Kratt, B. Chen, R. Měch, O. Deussen, and B. Benes, "Inverse Procedural Modelling of Trees," *Computer Graphics Forum*, vol. 33, no. 6, 2014.
[23] D. Ritchie, B. Mildenhall, N. D. Goodman, and P. Hanrahan, "Controlling Procedural Modeling Programs with Stochastically-ordered Sequential Monte Carlo," *ACM Transactions on Graphics*, vol. 34, no. 4, 2015.
[24] J. Talton, L. Yang, R. Kumar, M. Lim, N. Goodman, and R. Měch, "Learning Design Patterns with Bayesian Grammar Induction," in *Proc. UIST*, 2012.
[25] A. Yuille and D. Kersten, "Vision as Bayesian inference: analysis by synthesis?" *Trends in Cognitive Sciences*, pp. 301–308, 2006.
[26] O. Teboul, I. Kokkinos, L. Simon, P. Koutsourakis, and N. Paragios, "Shape Grammar Parsing via Reinforcement Learning," in *Proc. CVPR*, 2011.
[27] A. Martinovic and L. Van Gool, "Bayesian Grammar Learning for Inverse Procedural Modeling," in *Proc. CVPR*, 2013.
[28] T. D. Kulkarni, W. Whitney, P. Kohli, and J. B. Tenenbaum, "Deep convolutional inverse graphics network," in *Proc. NIPS*, 2015.
[29] L. Romaszko, C. K. I. Williams, P. Moreno, and P. Kohli, "Vision-as-inverse-graphics: Obtaining a rich 3d explanation of a scene from a single image," in *ICCV workshops*, 2017.
[30] H. Huang, E. Kalogerakis, E. Yumer, and R. Mech, "Shape Synthesis from Sketches via Procedural Models and Convolutional Networks," *IEEE Trans. Vis. & Comp. Graphics*, vol. 23, no. 8, 2017.
[31] D. Ritchie, A. Thomas, P. Hanrahan, and N. D. Goodman, "Neurally-Guided Procedural Models: Amortized Inference for Procedural Graphics Programs using Neural Networks," in *Proc. NIPS*, 2016.
[32] G. Nishida, I. Garcia-Dorado, D. G. Aliaga, B. Benes, and A. Bousseau, "Interactive Sketching of Urban Procedural Models," *ACM Transactions on Graphics*, vol. 35, no. 4, 2016.
[33] J. Wu and J. B. Tenenbaum, "Neural Scene De-rendering," in *Proc. CVPR*, 2017.
[34] S. M. A. Eslami, N. Heess, T. Weber, Y. Tassa, D. Szepesvari, K. Kavukcuoglu, and G. Hinton, "Attend, Infer, Repeat: Fast Scene Understanding with Generative Models," in *Proc. NIPS*, 2016.
[35] A. Neelakantan, Q. V. Le, and I. Sutskever, "Neural Programmer: Inducing Latent Programs with Gradient Descent," in *Proc. ICLR*, 2016.
[36] S. Reed and N. de Freitas, "Neural Programmer-Interpreters," in *Proc. ICLR*, 2016.

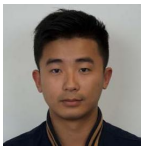
[37] M. Denil, S. Gómez Colmenarejo, S. Cabi, D. Saxton, and N. De Freitas, "Programmable Agents," *arXiv preprint arXiv:1706.06383*, 2017.
[38] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "DeepCoder: Learning to Write Programs," in *Proc. ICLR*, 2017.
[39] A. Joulin and T. Mikolov, "Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets," in *Proc. NIPS*, 2015.
[40] W. Zaremba and I. Sutskever, "Learning to Execute," *arXiv preprint arXiv:1410.4615*, 2014.
[41] W. Zaremba, T. Mikolov, A. Joulin, and R. Fergus, "Learning Simple Algorithms from Examples," in *Proc. ICML*, 2016.
[42] Ł. Kaiser and I. Sutskever, "Neural GPUs Learn Algorithms," in *Proc. ICLR*, 2016.
[43] C. Liang, J. Berant, Q. Le, K. D. Forbus, and N. Lao, "Neural Symbolic Machines: Learning Semantic Parsers on Freebase with Weak Supervision," in *Proc. ACL*, 2017.
[44] J. Johnson, B. Hariharan, L. Van Der Maaten, J. Hoffman, L. Fei-Fei, C. L. Zitnick, and R. Girshick, "Inferring and Executing Programs for Visual Reasoning," in *Proc. ICCV*, 2017.
[45] R. Hu, J. Andreas, M. Rohrbach, T. Darrell, and K. Saenko, "Learning to reason: End-to-end module networks for visual question answering," in *Proc. ICCV*, 2017.
[46] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein, "Neural Module Networks," in *Proc. CVPR*, 2016.
[47] Y. Tian, A. Luo, X. Sun, K. Ellis, W. T. Freeman, J. B. Tenenbaum, and J. Wu, "Learning to infer and execute 3d shape programs," in *ICLR*, 2019.
[48] K. Ellis, D. Ritchie, A. Solar-Lezama, and J. B. Tenenbaum, "Learning to Infer Graphics Programs from Hand-Drawn Images," *arXiv preprint arXiv:1707.09627*, 2017.
[49] S. Tulsiani, H. Su, L. J. Guibas, A. A. Efros, and J. Malik, "Learning Shape Abstractions by Assembling Volumetric Primitives," in *Proc. CVPR*, 2017.
[50] C. Zou, E. Yumer, J. Yang, D. Ceylan, and D. Hoiem, "3D-PRNN: Generating Shape Primitives with Recurrent Neural Networks," in *Proc. ICCV*, 2017.
[51] L. Li, M. Sung, A. Dubrovina, L. Yi, and L. Guibas, "Supervised fitting of geometric primitives to 3d point clouds," *CVPR*, pp. 2647–2655, 2019.
[52] D. Paschalidou, A. O. Ulusoy, and A. Geiger, "Superquadrics revisited: Learning 3d shape parsing beyond cuboids," *CoRR*, vol. abs/1904.09970, 2019.
[53] K. Genova, F. Cole, A. Sud, A. Sarna, and T. Funkhouser, "Local deep implicit functions for 3d shape," in *CVPR*, June 2020.
[54] J. Huang, J. Gao, V. Ganapathi-Subramanian, H. Su, Y. Liu, C. Tang, and L. J. Guibas, "Deepprimitive: Image decomposition by layered primitive detection," *Computational Visual Media*, vol. 4, no. 4, pp. 385–397, Dec 2018.
[55] J. Gao, C. Tang, V. Ganapathi-Subramanian, J. Huang, H. Su, and L. J. Guibas, "Deepspline: Data-driven reconstruction of parametric curves and surfaces," *CoRR*, vol. abs/1901.03781, 2019.
[56] Z. Chen, A. Tagliasacchi, and H. Zhang, "Bsp-net: Generating compact meshes via binary space partitioning," *CVPR*, 2020.
[57] B. Deng, K. Genova, S. Yazdani, S. Bouaziz, G. Hinton, and A. Tagliasacchi, "Cvxnet: Learnable convex decomposition," in *CVPR*, 2020, pp. 31–41.
[58] T. Deprelle, T. Groueix, M. Fisher, V. Kim, B. Russell, and M. Aubry, "Learning elementary structures for 3d shape generation and matching," in *Proc. NeurIPS*, vol. 32, 2019.
[59] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.
[60] J. E. Hopcroft, R. Motwani, and U. J. D, *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2001.
[61] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.
[62] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy Gradient Methods for Reinforcement Learning with Function Approximation," in *Proc. NIPS*, 1999.
[63] Z. Lun, E. Kalogerakis, and A. Sheffer, "Elements of style: Learning perceptual shape style similarity," *ACM Transactions on Graphics*, vol. 34, no. 4, 2015.
[64] M. J. D. Powell, "An efficient method for finding the minimum of a function of several variables without calculating derivatives," *The Computer Journal*, vol. 7, no. 2, p. 155, 1964.
[65] "Trimble 3D Warehouse," https://3dwarehouse.sketchup.com/.
[66] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.
[67] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman, "Return of the devil in the details: Delving deep into convolutional nets," in *Proc. BMVC*, 2014.
[68] R. O. Duda and P. E. Hart, "Use of the hough transformation to detect lines and curves in pictures," *Commun. ACM*, vol. 15, no. 1, pp. 11–15, Jan. 1972.
[69] C. Choy, J. Gwak, and S. Savarese, "4d spatio-temporal convnets: Minkowski convolutional neural networks," in *CVPR*, 2019.
[70] A. X. Chang, T. A. Funkhouser, L. J. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, J. Xiao, L. Yi, and F. Yu, "Shapenet: An information-rich 3d model repository," *CoRR*, vol. abs/1512.03012, 2015.
[71] S. Koch, A. Matveev, Z. Jiang, F. Williams, A. Artemov, E. Burnaev, M. Alexa, D. Zorin, and D. Panozzo, "Abc: A big cad model dataset for geometric deep learning," in *CVPR*, 2019.

6 Biography

Biographies of the authors:

Gopal Sharma received B.Tech. in Electrical Engineering from IIT-Roorkee, India in 2016. He received an MS degree in computer science from University of Massachusetts Amherst in 2019. Presently, he is a doctoral candidate in computer science at University of Massachusetts Amherst.

Rishabh Goyal completed his undergraduate studies in Computer Science and Engineering at the IIT Kanpur. Following this he spent an year working at IBM Research Labs, developing artificial conversational agents. He is currently pursuing a masters in Computer Science at the University of Illinois at Urbana-Champaign.



Difan Liu received the BS degree from the University of Science and Technology of China in 2017, and the MS degree from University of Massachusetts Amherst in 2020. He is working toward his PhD degree in computer science at University of Massachusetts Amherst.

Evangelos Kalogerakis is an Associate Professor at the College of Information and Computer Sciences at University of Massachusetts Amherst, which he joined in 2012. He was a postdoc at Stanford University (2010-2012). He earned his PhD from the University of Toronto in 2010.

Subhransu Maji is an Associate Professor in the College of Information and Computer Sciences at the University of Massachusetts, Amherst. He obtained his Ph.D. from the University of California at Berkeley in 2011, and a B.Tech. in Computer Science and Engineering from IIT Kanpur in 2006