CSGNet: Neural Shape Parser for Constructive Solid Geometry

Gopal Sharma  Rishabh Goyal  Difan Liu  Evangelos Kalogerakis  Subhransu Maji
University of Massachusetts, Amherst
{gopalsharma,risgoyal,dliu,kalo,smaji}@cs.umass.edu

Abstract

We present a neural architecture that takes as input a 2D or 3D shape and outputs a program that generates the shape. The instructions in our program are based on constructive solid geometry principles, i.e., a set of boolean operations on shape primitives defined recursively. Bottom-up techniques for this shape parsing task rely on primitive detection and are inherently slow since the search space over possible primitive combinations is large. In contrast, our model uses a recurrent neural network that parses the input shape in a top-down manner, which is significantly faster and yields a compact and easy-to-interpret sequence of modeling instructions. Our model is also more effective as a shape detector compared to existing state-of-the-art detection techniques. We finally demonstrate that our network can be trained on novel datasets without ground-truth program annotations through policy gradient techniques.

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 [28, 9, 12, 16]. 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 (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. The reason for choosing higher-level primitives is not incidental. Describing shapes with as few as possible primitives and operations is highly desirable for designers since it is compact, makes subsequent editing easier, and is perhaps better at capturing aspects of human shape perception such as view invariance, compositionality, and symmetry [5].

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) modeling [29] as seen in Figure 1. This 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 operations matters. Figure 1 demonstrates an example where a complex object is created through boolean operations that combine simpler objects. If one performs a small change e.g., swap two operations, the resulting object becomes entirely different. From this aspect, the shape modeling process could be thought of as a visual program i.e., an ordered set of modeling instructions. Finally, a challenge is that we would like to learn an efficient parser that generates a compact program (e.g., with the fewest instructions) without relying on a vast number of shapes annotated with their programs for a target domain.
To tackle these challenges we designed a memory-enabled network architecture, that given a target 2D image of a shape, or a target 3D shape, generates a CSG program to generate it. To train our network we created a large synthetic dataset of automatically generated 2D and 3D programs. Networks trained on this dataset however lead 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 [44]. Combining our parser with a CSG rendering engine allows the parser to receive feedback based on the visual difference between the target shape and generated shape. Thus the parser network can be trained to minimize this difference.

Our contributions are as follows. First we show that the proposed architecture is efficient and effective at inferring CSG programs for 2D and 3D shapes across a number of domains. Second we show that the parser can be learned using reinforcement learning techniques on novel datasets without program annotations. Third, we show that the parser is a better and faster shape detector than state-of-the-art detection approaches that only rely on bottom-up cues. We conjecture that this is because the parser jointly reasons about presence and ordering during parsing unlike the detector.

2. Related Work

Our work is primarily related to neural program induction methods. Secondly, it is also related to “vision-as-inverse-graphics” approaches, as well as neural network-based methods that predict shape primitives or parameters of procedural graphics models. Below, we briefly overview these prior methods, and explain differences from our work.

Neural program induction. Our method is inspired by recent progress in neural network-based methods that infer programs expressed in some high-level language to solve a task. These methods often employ variants of recurrent neural networks whose parameters are trained to predict desired program outputs given exemplar inputs, such as answers to questions involving complex arithmetic, logical, or semantic parsing operations [32, 35, 11, 4, 23, 49, 48, 24, 30].

In the context of visual reasoning, several authors [22, 20] proposed architectures that produce programs composed of functions that perform compositional reasoning on the input image. They also incorporate an execution engine that produces the result of the program through a neural module network [3]. In contrast our method aims to produce a generative program consisting of shape modeling functions that match a target image.

Vision-as-inverse-graphics. A well-known approach to visual analysis is to generate and fit hypotheses of scenes or objects to input image data i.e., perform analysis-by-synthesis [47]. Kulkani et al. [27] proposed sampling-based probabilistic inference to estimate parameters of stochastic graphics models (e.g., human body parameters, or parameters of rotationally symmetric objects) representing the space of hypothesized scenes given an input image. Shape grammars (or so-called inverse procedural modeling techniques) have alternatively been used in analysis-by-synthesis image parsing frameworks [42, 31], yet they have the disadvantage of not modeling long-range dependencies in the parsing task, and are often specific to a particular shape class (e.g., buildings). More recent approaches employ Convolutional Neural Network (CNN) to infer parameters of objects [28] or whole scenes [39]. A similar trend is observed in graphics applications where CNNs are used to map input images or partial shapes to procedural model parameters [21, 38, 33]. Wu et al. [45] 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. [15] uses 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.

In contrast, we do not aim at parsing images or scenes into a collection of objects and their parameters. We instead parse input images or 3D shapes into a sequence of modeling operations on primitives (i.e., a visual program) to match a target image. In our setting, the space of outputs is much larger and the order of operations in our visual programs matter. To deal with this complexity, we use a combination of supervised pretraining, reinforcement learning, reward design, and post-optimization of modeling parameters, described in the next Section.

Neural primitive fitting. Tulsiani et al. [43] proposed a volumetric convolutional network architecture that predicts a fixed number of cuboidal primitives to describe an input 3D shape. To better handle a variable number of primitives, Zou et al. [50] instead proposed an LSTM-based architecture that predicts boxes given input depth images. We also aim at deriving geometrically interpretable explanations of shapes in terms of primitives. However, our network is not limited to predicting a single type of primitives (e.g., cubes), but also outputs modeling operations acting on them, or in other words supports a significantly richer modeling paradigm. The program can be used not only to geometrically describe the input shape but can also be directly edited to manipulate it if desired. Finally, Ellis et al. [14] proposed a neural network architecture to extract various hand-drawn primitives (lines, circles, rectangles) in images, which are then grouped into LaTeX programs. Their program synthesis is posed as a constraint satisfaction problem.
3. Designing a Neural Shape Parser

In this section, we first present our neural shape parser that can induce programs for 2D/3D shapes. The goal of the parser $\pi$ 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 $\Theta$ 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) [10] that have been widely used for sequential prediction tasks such as generating natural language and speech. The overall network can be written as $\pi(I) = \Theta \circ \Phi(I)$. The space of programs can be efficiently described according to a context-free grammar [19]. For example, in constructive solid geometry the instructions consist of drawing primitives (e.g., spheres, cubes, cylinders, etc.) and performing boolean operations described as a grammar with the following production rules:

$$
S \rightarrow E \\
E \rightarrow E E T \mid P \\
T \rightarrow OP_1|OP_2|\ldots|OP_m \\
P \rightarrow SHAPE_1|SHAPE_2|\ldots|SHAPE_n
$$

Each rule indicates possible derivations of a non-terminal symbol separated by the $|$ symbol. Here $S$ is the start symbol, $OP_i$ is chosen from a set of defined modeling operations and the $SHAPE_i$ 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., $SHAPE_1SHAPE_2OP_1SHAPE_3OP_2$. Figure 2 also gives an example of a program predicted by the network, that follows the grammar described above.

3.1. Learning

Given an input $I$ the parser network $\pi$ 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 hence the network must also learn to generate grammatical programs.
**Supervised learning:** When target programs are available the architecture can be trained with standard supervised learning techniques. Training data in this case consists of shape and program pairs \((I^i, P^i)\), \(i = 1, \ldots, N\). In our implementation, the RNN produces a categorical distribution \(\pi_\theta\) over instructions \(a \in A\) at every time step. Similarly the ground-truth program \(P^i\) can be written as sequence of instructions \(g_1^i, g_2^i \ldots g_{T_i}^i\), where \(T_i\) is the length of the program \(P^i\). The parameters \(\theta\) can be learned to maximize the log-likelihood of the ground truth instructions:

\[
L(\theta) = \sum_{i=1}^{N} \sum_{t=1}^{T_i} \log \pi_\theta(g_t^i|g_{1:t-1}, I^i).
\]  

(1)

**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 [44] from the reinforcement learning (RL) literature can instead be used in this case.

Concretely, the parser \(\pi_\theta\), that represents a policy network, can be used to sample a program \(y = (a_1, a_2 \ldots a_T)\) conditioned on the input shape \(I\). 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 \(\theta\) that maximize the expected rewards over programs sampled under the predicted distribution \(\pi_\theta(I)\) across images \(I\) sampled from a distribution \(D\):

\[
E_{I \sim D} \left[ J_\theta(I) \right] = E_{I \sim D}E_{y \sim \pi_\theta(I)} \left[ R \right].
\]

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:

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

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 popular REINFORCE [44] algorithm computes a Monte-Carlo estimate as:

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

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

\[
\nabla_\theta J_\theta(I) = \frac{1}{S} \sum_{s=1}^{S} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\hat{a}_t^s|\hat{a}_{1:t-1}^s, I)(R^s - b).
\]

(2)

A good choice of the baseline is the expected value of returns starting from \(t\) [40, 44]. We compute 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 edges of each shape. The CD is between two point sets, \(x\) and \(y\), is defined as follows:

\[
Ch(x, y) = \frac{1}{2|\text{x}|} \sum_{x \in \text{x}} \min_{y \in \text{y}} \|x - y\|_2^2 + \frac{1}{2|\text{y}|} \sum_{y \in \text{y}} \min_{x \in \text{x}} \|x - y\|_2^2.
\]

The points are scaled by the image diagonal, thus \(Ch(x, y) \in [0, 1] \ \forall x, y\). 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:

\[
R = \begin{cases} 
  f \left( Ch(\text{Edge}(I), \text{Edge}(Z(y))) \right), & y \text{ is valid} \\
  0, & y \text{ is invalid}.
\end{cases}
\]

where \(f\) is a shaping function and \(Z\) is the CSG rendering engine. Since invalid programs get zero reward, the maximum length constraint on the programs encourages 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 \(\gamma\) encourages CD close to zero. We found that \(\gamma = 20\) provides a good trade-off between program length and visual similarity.

**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. Our parser produces 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 [34] to optimize the parameters using feedback from the rendering engine. On our dataset where shapes have up to 7 primitives, the search space is relatively small and the algorithm converges to a local minima in about 10 iterations and consistently improves the results.

4. Experiments

We describe our experiments on different datasets exploring the generalization capabilities of our network (CSGNet). 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. We discuss our qualitative and quantitative results on the above datasets.

4.1. Datasets

![Figure 3. Samples created from our synthetically generated programs. 2D samples in top row and 3D samples in bottom row.](image)

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:

\[
S \rightarrow E; \ E \rightarrow EET; \\
E \rightarrow sp(L, R) | cu(L, R) | cy(L, R, H) \\
T \rightarrow \text{intersect} | \text{union} | \text{subtract}; \\
P \rightarrow \text{square} | \text{circle} | \text{triangle}; \\
L \rightarrow [8 : 8 : 56]^2; \ R \rightarrow [8 : 4 : 32].
\]

Primitives are specified by their type: *square, circle, or triangle*, locations \(L\) and circumscribing circle of radius \(R\) on a canvas of size \(64 \times 64\). 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 \(64 \times 64\) pixels. Samples from our dataset are shown in Figure 3. Table 1 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:

\[
S \rightarrow E; \ E \rightarrow EET; \\
E \rightarrow sp(L, R) | cu(L, R) | cy(L, R, H) \\
T \rightarrow \text{intersect} | \text{union} | \text{subtract}; \\
P \rightarrow \text{square} | \text{circle} | \text{triangle}; \\
L \rightarrow [8 : 8 : 56]^3; \ R \rightarrow [8 : 4 : 32]; \ H \rightarrow [8 : 4 : 32].
\]

The same three binary operations are used as in the 2D case. Three basic solids are denoted by ‘sp’: Sphere, ‘cu’: Cube, ‘cy’: Cylinder. \(L\) represents the center of primitive in 3D voxel grid. \(R\) specifies radius of sphere and cylinder, and also specifies size of cube. \(H\) is the height of cylinder. The primitives are rendered as voxel grids and the programs are executed on a 3D volumetric grid of size \(64 \times 64 \times 64\). We used the same random sampling method as described for the synthetic 2D dataset, resulting in 3D CSG programs. 3D shape samples from this dataset are shown in Figure 3.

**2D CAD shapes.** We collected \(8K\) CAD shapes from the Trimble 3D Warehouse dataset [2] in three categories: chair, desk and lamps. We rendered the CAD shapes into \(64 \times 64\) 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>2D</th>
<th>3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>Train</td>
<td>Val</td>
</tr>
<tr>
<td>3</td>
<td>25k</td>
<td>5k</td>
</tr>
<tr>
<td>5</td>
<td>100k</td>
<td>10k</td>
</tr>
<tr>
<td>7</td>
<td>150k</td>
<td>20k</td>
</tr>
<tr>
<td>9</td>
<td>250k</td>
<td>20k</td>
</tr>
<tr>
<td>11</td>
<td>350k</td>
<td>20k</td>
</tr>
<tr>
<td>13</td>
<td>350k</td>
<td>20k</td>
</tr>
</tbody>
</table>

Table 1. Statistics of our 2D and 3D synthetic dataset.
4.2. Implementation details

The input 2D or 3D shape $I$ is represented as pixel and voxel occupancy grid respectively. Our encoder is based on an image-based convnet in the case of 2D inputs, and a volumetric convnet in the case of 3D inputs. The output of the encoder $\Phi(I)$ 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. For the 3D CSG 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 rendered from programs of variable length (up to 13 for 2D and up to 7 for 3D dataset) from training dataset. 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 [25] 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. Our implementation is based on PyTorch [1]. Our source code and datasets are available on our project page: https://hippogriff.github.io/CSGNet.

4.3. Results

We evaluate our network, called CSGNet, in two different ways: (i) as a model for inferring the entire program, and (ii) as model for inferring primitives, i.e., as an object detector.

4.3.1 Inferring programs

Evaluation on the synthetic 2D shapes. We perform supervised learning to train 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. Table 2 compares CSGNet to this NN baseline using the Chamfer distance between the test target and predicted shapes. 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 4 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.

<table>
<thead>
<tr>
<th>Method</th>
<th>NN $k=1$</th>
<th>CSGNet $k=1$</th>
<th>CSGNet $k=5$</th>
<th>CSGNet $k=10$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>1.94</td>
<td>1.69</td>
<td>1.46</td>
<td>1.39</td>
</tr>
</tbody>
</table>

Table 2. Comparison of the supervised network (CSGNet) with the NN baseline on the synthetic 2D dataset. Results are shown using Chamfer Distance (CD) metric by varying beam sizes (k) during decoding. CD is in number of pixels.

![Figure 4. Comparison of performance on synthetic 2D dataset.](image)

Figure 4. Comparison of performance on synthetic 2D dataset. a) Input image, b) NN-retrieved image, c) top-1 prediction, and d) best result from top-10 beam search predictions of CSGNet.

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 2D CAD dataset using policy gradients.

Table 3 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 CSGNet trained in a supervised manner only on our synthetic dataset. With beam search, the performance of this variant improves compared to NN. Most importantly, 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. 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 5 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.

**Logos.** Here, 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 6. 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 train a 3D-CNN + GRU (3D-CSGNet) network on the 3D CSG synthetic dataset explained in Section 4.1. The input to our 3D-CSGNet are voxelized shapes in a 64×64×64 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 7 show pairs of input voxel grids and our output shapes from the test split of the 3D dataset. The qualitative results are shown in the Table 4, where we compare our 3D-CSGNet at different beam search decodings with NN method. The results indicate that our method is promising in inducing correct programs, which also have the advantage of accurately reconstructing the voxelized surfaces into high-resolution surfaces.

### Table 3. Comparison of various approaches on the CAD shape dataset.

<table>
<thead>
<tr>
<th>Method</th>
<th>Train</th>
<th>Test</th>
<th>CD (@refinement iterations)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>i=0</td>
</tr>
<tr>
<td>NN</td>
<td>-</td>
<td>-</td>
<td>1.92</td>
</tr>
<tr>
<td>CSGNet</td>
<td>Super</td>
<td>k=1</td>
<td>2.30</td>
</tr>
<tr>
<td></td>
<td>Super</td>
<td>k=10</td>
<td>1.60</td>
</tr>
<tr>
<td>CSGNet</td>
<td>RL</td>
<td>k=1</td>
<td>1.26</td>
</tr>
<tr>
<td></td>
<td>RL</td>
<td>k=10</td>
<td>1.14</td>
</tr>
</tbody>
</table>

Figure 5. Comparison of performance on the 2D CAD dataset. From left column to right column: a) Input image, b) NN retrieved image, c) top-1 prediction from CSGNet in the supervised learning mode, d) top-1 prediction from CSGNet fine-tuned with RL (policy gradient), e) best result from beam search from CSGNet fine-tuned with RL, f) refining our results using the visually guided search on the best beam result (“full” version of our method).

Figure 6. 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.

### 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 them-
selves). This allows us to directly compare our approach with bottom-up object detection techniques.

In particular, we compare against a state-of-the-art object detector (Faster R-CNNs [36]). The Faster R-CNN is based on the VGG-M network [8] 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 [13] 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.

Table 4. Comparison of the supervised network (3D-CSGNet) with NN baseline on 3D dataset. Results are shown using IOU(%) metric by varying beam sizes \((k)\) during decoding.

<table>
<thead>
<tr>
<th>Method</th>
<th>NN</th>
<th>3D-CSGNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOU (%)</td>
<td></td>
<td>( k=1)</td>
</tr>
<tr>
<td></td>
<td>73.2</td>
<td>80.1</td>
</tr>
</tbody>
</table>

Table 5. MAP of detectors on the synthetic 2D shape dataset. We also report detection speed measured as images/second on a NVIDIA 1070 GPU.

<table>
<thead>
<tr>
<th>Method</th>
<th>Circle</th>
<th>Square</th>
<th>Triangle</th>
<th>Mean</th>
<th>Speed (im/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faster R-CNN</td>
<td>87.4</td>
<td>71.0</td>
<td>81.8</td>
<td>80.1</td>
<td>5</td>
</tr>
<tr>
<td>CSGNet, ( k = 10)</td>
<td>86.7</td>
<td>79.3</td>
<td>83.1</td>
<td>83.0</td>
<td>80</td>
</tr>
<tr>
<td>CSGNet, ( k = 40)</td>
<td><strong>88.1</strong></td>
<td><strong>80.7</strong></td>
<td><strong>84.1</strong></td>
<td><strong>84.3</strong></td>
<td><strong>20</strong></td>
</tr>
</tbody>
</table>

5. Conclusion

We believe that our work represents a first step towards the automatic generation of modeling programs given target visual content, which we believe is quite ambitious and hard problem. We demonstrated results of generated programs in various domains, including logos, 2D binary shapes, and 3D CAD shapes, as well as an analysis-by-synthesis application in the context of 2D shape primitive detection.

One might argue that the 2D images and 3D shapes our method parsed are relatively simple in structure or geometry. However, we would also 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 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, a challenging research direction would be 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. Other promising directions would be to explore how to combine bottom-up proposals and top-down approaches for parsing shapes, in addition to exploring top-down program generation strategies.

Acknowledgments. We acknowledge support from NSF (CHS-1422441, CHS-1617333, IIS-1617917) and the MassTech Collaborative grant for funding the UMass GPU cluster.
References