

Inferring Graphics Programs from Images

Anonymous Author(s)

Affiliation

Address

email

Abstract

1

2 1 Introduction

3 How could an agent go from noisy, high-dimensional perceptual input to a symbolic, abstract object,
 4 like a computer program? Here we consider this problem within a graphics program synthesis domain.
 5 We develop an approach for converting natural images, such as hand drawings, into executable source
 6 code for drawing the original image. The graphics programs in our domain draw simple figures like
 7 those found in machine learning papers (see Fig.1). [The use of ‘graphics programs / visual programs’
 8 in the paper title, title of this section, and the body of this section feels too broad. ‘Graphics program’
 9 could conjur a lot of different ideas (esp. 3D graphics); don’t want to set the reader up to expect one
 10 thing and then be disappointed that what you’ve done isn’t that. You bring up diagram-drawing later
 11 in the intro; I think it should be made clear sooner (and certainly mentioned explicitly in the abstract,
 12 when you get around to writing that).]

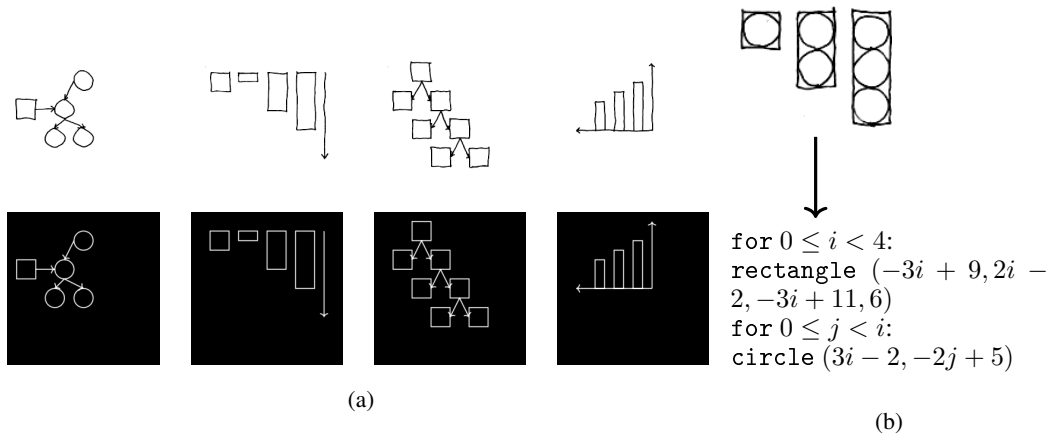


Figure 1: (a): Model learns to convert hand drawings (top) into LAT\textsubscript{E}X (bottom). (b) Synthesizes high-level *graphics program* from hand drawing.

13 High dimensional perceptual input may seem ill matched to the abstract semantics of a programming
 14 language. But programs with constructs like recursion or iteration produce a simpler *execution trace*
 15 of primitive actions; for our domain the primitive actions are drawing commands. Our hypothesis is
 16 that the execution trace of the program is better aligned with the perceptual input, and that the trace
 17 can act as a kind of bridge between perception and programs. We test this hypothesis by developing
 18 a model that learns to map from an image to the execution trace of the graphics program that drew
 19 it. With the execution trace in hand, we can bring to bear techniques from the program synthesis

community to recover the latent graphics program. This family of techniques, called *constraint-based program synthesis* [?], work by modeling a set of possible programs inside of a constraint solver, like a SAT or SMT solver [?].

We develop a hybrid architecture for inferring graphics programs. Our approach uses a deep neural network infer an execution trace from an image; this network recovers primitive drawing operations such as lines, circles, or arrows, along with their parameters. For added robustness, we use the deep network as a proposal distribution for a stochastic search over execution traces. Finally, we use techniques in the program synthesis community to recover the program from its trace. The program synthesizer discovers constructs like loops and geometric operations like reflections and affine transformations. [This paragraph is all about making things a bit more specific, so you really need more specifics about program synth here.]

Each of these three components – the deep network, the stochastic search, the program synthesizer – confers its own advantages. From the deep network, we get a fast system that can recover plausible execution traces in about a minute [A minute seems slow to me, for deep net inference. Are you talking about training time, here, or...?]. From the stochastic search we get added robustness; essentially, the stochastic search can correct mistakes made by the deep network’s proposals. From the program synthesizer, we get abstraction: our system recovers coordinate transformations, for loops, and subroutines, which are useful for downstream tasks and can help correct some mistakes of the earlier stages. [I wonder if this would work even better as a bulleted list...]

2 Related work

attend infer repeat: [1]. Crucial distinction is that they focus on learning the generative model jointly with the inference network. Advantages of our system is that we learn symbolic programs, and that we do it from hand sketches rather than synthetic renderings.

ngpm: [2]. We build on the idea of a guide program, extending it to scenes composed of objects, and then show how to learn programs from the objects we discover.

Sketch-n-Sketch: [3]. Semiautomated synthesis presented in a nice user interface. Complementary to our work: you could pass a sketch to our system and then pass the program to sketch-n-sketch

Converting hand drawings into procedural models using deep networks: [4, 5].

3 Neural architecture for inferring image parses

We developed a deep network architecture for efficiently inferring a execution trace, T , from an image, I . Our model constructs the trace one drawing command at a time. When predicting the next drawing command, the network takes as input the target image I as well as the rendered output of previous drawing commands. Intuitively, the network looks at the image it wants to explain, as well as what it has already drawn. It then decides either to stop drawing or proposes another drawing command to add to the execution trace; if it decides to continue drawing, the predicted primitive is rendered to its “canvas” and the process repeats.

Figure 2 illustrates this architecture. We first pass a 256×256 target image and a rendering of the trace so far to a convolutional network – these two inputs are represented as separate channels for the convnet. Given the features extracted by the convnet, a multilayer perceptron then predicts a distribution over the next drawing command to add to the trace. We predict the drawing command token-by-token, and condition each token both on the image features and on the previously generated tokens. For example, the network first decides to emit the `circle` token conditioned on the image features, then it emits the x coordinate of the circle conditioned on the image features and the `circle` token, and finally it predicts the y coordinate of the circle conditioned on the image features, the `circle` token, and the x coordinate. [There are some more details that are important to provide about this architecture, though possibly in an Appendix: the functional form(s) of the probability distributions over tokens, the network layer sizes, which MLPs share parameters, etc.]

[Planning to move the description of SMC / beam search up here, too?]

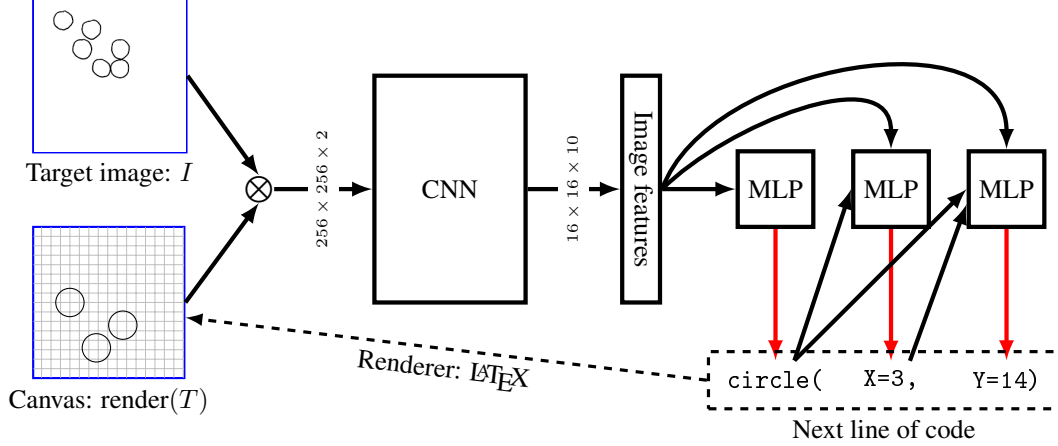


Figure 2: Our neural architecture for inferring the execution trace of a graphics program from its output. **Blue**: network inputs. **Black**: network operations. **Red**: samples from a multinomial. Typewriter font: network outputs. Renders snapped to a 16×16 grid, illustrated in gray. [Thoughts on improving this figure: (1) Convnet diagrams typically show the sequence of layers, if possible (space might not permit it here, but those thin arrows just aren’t doing it for me). (2) Are the target image / canvas convolved down independently, or jointly (i.e. starting as a 2-channel image)? That’s an important detail that’s not clear with the current figure/explanation. (3) The three circles downstream from ‘Image Features’ are supposed to be MLPs, I assume(?), but it took me a little while to parse that. Having some visual way of clearly separating network operations from data (color, perhaps) would go a long way.]

68 The distribution over the next drawing command factorizes:

$$\mathbb{P}_\theta[t_1 t_2 \cdots t_K | I, T] = \prod_{k=1}^K \mathbb{P}_\theta[t_k | f_\theta(I, \text{render}(T)), \{t_j\}_{j=1}^{k-1}] \quad (1)$$

69 where $t_1 t_2 \cdots t_K$ are the tokens in the drawing command, I is the target image, T is an execution trace,
70 θ are the parameters of the neural network, and $f_\theta(\cdot, \cdot)$ is the image feature extractor (convolutional
71 network). The distribution over execution traces factorizes as:

$$\mathbb{P}_\theta[T | I] = \prod_{n=1}^{|T|} \mathbb{P}_\theta[T_n | I, T_{1:(n-1)}] \times \mathbb{P}_\theta[\text{STOP} | I, T] \quad (2)$$

72 where $|T|$ is the length of execution trace T , and the STOP token is emitted by the network to signal
73 that the execution trace explains the image.

74 We train the network by sampling execution traces T and target images I for randomly generated
75 scenes, and maximizing (2) wrt θ by gradient ascent. Training does not require backpropagation across
76 the entire sequence of drawing commands: drawing to the canvas ‘blocks’ the gradients, effectively
77 offloading memory to an external visual store. In a sense, this model is like an autoregressive variant
78 of AIR [1] without attention.

79 [I like that you make this connection, but it could be made more precisely. Specifically, (1) the
80 architecture isn’t *really* recurrent (it uses no hidden state cells), so it’d be good to use a different term
81 or drop this part of the point: (2) training of recurrent nets is also typically fully-supervised (Most
82 RNNs lack latent variables per timestep)—if you’re thinking about AIR specifically, maybe just say
83 that, and (3) it’s like an autoregressive AIR *without attention*.] [Something related to this that’s also
84 cool to point out: training this model doesn’t require backpropagation across the entire sequence of
85 drawing commands (drawing to the canvas ‘blocks’ the gradients, effectively offloading memory to
86 an external (visual) store, so in principle it might be scalable to much longer sequences.]

87 This network suffices to “derender” images like those shown in Figure ?? . We can perform a beam
88 search decoding to recover what the network thinks is the most likely execution trace for images
89 like these. But, if the network makes a mistake (predicts an incorrect line of code), it has no way

<code>circle(x, y)</code>	Circle at (x, y)
<code>rectangle(x_1, y_1, x_2, y_2)</code>	Rectangle with corners at (x_1, y_1) & (x_2, y_2)
<code>LINE(x_1, y_1, x_2, y_2, arrow $\in \{0, 1\}$, dashed $\in \{0, 1\}$)</code>	Line from (x_1, y_1) to (x_2, y_2) , optionally with an arrow and/or dashed
<code>STOP</code>	Finishes execution trace inference

Table 1: The deep network in (2) predicts drawing commands, shown above.

of recovering from the error. In order to derender an image with n objects, it must correctly predict n drawing commands – so its probability of success will decrease exponentially in n , assuming it has any nonzero chance of making a mistake. For added robustness as n becomes large, we treat the neural network outputs as proposals for a SMC sampling scheme. For the SMC sampler, we use pixel wise distance as a surrogate for a likelihood function; see supplement. Figure 3 compares the neural network with SMC against the neural network by itself or SMC by itself. Only the combination of the two passes a critical test of generalization: when trained on images with ≤ 8 objects, it successfully parses scenes with many more objects than the training data.

3.1 Generalizing to hand drawings

We believe that converting synthetic, noiseless images into a restricted subset of \LaTeX has limited usefulness. A more practical application is one that extends to hand drawings. We train the model to generalize to hand drawings by introducing noise into the renderings of the training target images. We designed this noise process to introduce the kinds of variations found in hand drawings (figure 4). We drew 100 figures by hand; see figure ?? . These were drawn reasonably carefully but not perfectly. Because our model assumes that objects are snapped to a 16×16 grid, we made the drawings on graph paper.

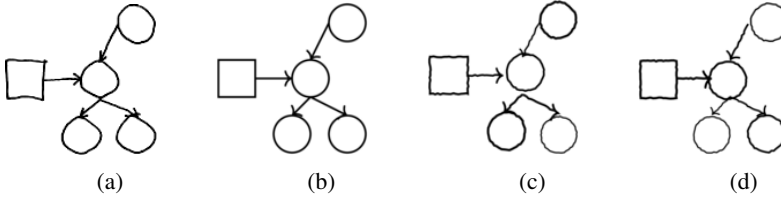


Figure 4: (a): a hand drawing. (b): Rendering of the parse our model infers for (a). We can generalize to hand drawings like these because we train the model on images corrupted by a noise process designed to resemble the kind of noise introduced by hand drawings - see (c) & (d) for noisy renderings of (b).

4 Synthesizing graphics programs from execution traces

```

Program → Command; ...; Command
Command → circle(Expression, Expression)
Command → rectangle(Expression, Expression, Expression, Expression)
Command → LINE(Expression, Expression, Expression, Expression, Boolean, Boolean)
Command → for( $0 \leq \text{Var} < \text{Expression}$ ) { Program }
Command → REFLECT(Axis) { Program }
Expression → Z * Var + Z
  Var → A free (unused) variable
  Z → an integer
  Axis → X = Z
  Axis → Y = Z

```

Table 2: Grammar over graphics programs. We allow loops (for), vertical/horizontal reflections (REFLECT), and affine transformations ($Z * \text{Var} + Z$).

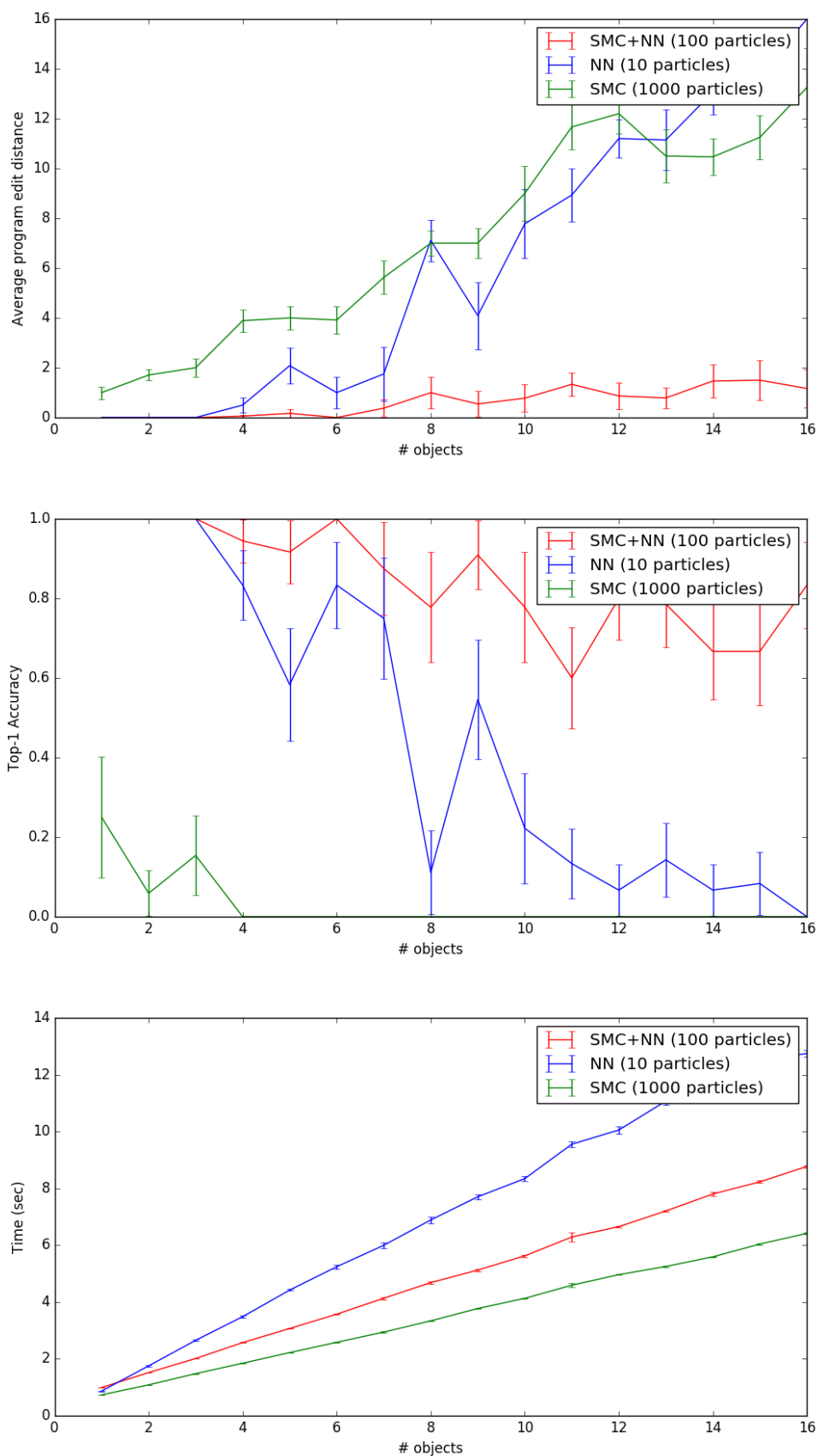


Figure 3: Using the model to parse latex output. The model is trained on diagrams with up to 8 objects. As shown above it generalizes to scenes with many more objects. Neither the stochastic search nor the neural network are sufficient on their own.

107 5 Neural networks for guiding SMC

108 Let $L(\cdot|\cdot) : \text{image}^2 \rightarrow \mathcal{R}$ be our likelihood function: it takes two images, an observed target image
 109 and a hypothesized program output, and gives the likelihood of the observed image conditioned on
 110 the program output. We want to sample from:

$$\mathbb{P}[p|x] \propto L(x|\text{render}(p))\mathbb{P}[p] \quad (3)$$

111 where $\mathbb{P}[p]$ is the prior probability of program p , and x is the observed image.

112 Let p be a program with L lines, which we will write as $p = (p_1, p_2, \dots, p_L)$. Assume the prior
 113 factors into:

$$\mathbb{P}[p] \propto \prod_{l \leq L} \mathbb{P}[p_l] \quad (4)$$

114 Define the distribution $q_L(\cdot)$, which happens to be proportional to the above posterior:

$$q_L(p_1, p_2, \dots, p_{L-1}, p_L) \propto q_{L-1}(p_1, p_2, \dots, p_{L-1}) \times \frac{L(x|\text{render}(p_1, p_2, \dots, p_{L-1}, p_L))}{L(x|\text{render}(p_1, p_2, \dots, p_{L-1}))} \times \mathbb{P}[p_L] \quad (5)$$

115 Now suppose we have some samples from $q_{L-1}(\cdot)$, and that we then sample a p_L from a distribu-
 116 tion proportional to $\frac{L(x|\text{render}(p_1, p_2, \dots, p_{L-1}, p_L))}{L(x|\text{render}(p_1, p_2, \dots, p_{L-1}))} \times \mathbb{P}[p_L]$. The resulting programs p are distributed
 117 according to q_L , and so are also distributed according to $\mathbb{P}[p|x]$.

118 How do we sample p_L from a distribution proportional to $\frac{L(x|\text{render}(p_1, p_2, \dots, p_{L-1}, p_L))}{L(x|\text{render}(p_1, p_2, \dots, p_{L-1}))} \times \mathbb{P}[p_L]$? We
 119 have a neural network that takes as input the target image x and the program so far, and produces
 120 a distribution over next lines of code (p_L). We write $\text{NN}(p_L|p_1, \dots, p_{L-1}; x)$ for the distribution
 121 output by the neural network. So we can sample from NN and then weight the samples by:

$$w(p_L) = \frac{\mathbb{P}[p_L]}{\text{NN}(p_L|p_1, \dots, p_{L-1}; x)} \times \frac{L(x|\text{render}(p_1, p_2, \dots, p_{L-1}, p_L))}{L(x|\text{render}(p_1, p_2, \dots, p_{L-1}))} \quad (6)$$

122 Then we can resample from these now weighted samples to get a new population of particles (here
 123 programs are particles), where each program now has L lines instead of $L - 1$.

124 This procedure can be seen as a particle filter, where each successive latent variable is another line of
 125 code, and the emission probabilities are successive ratios of likelihoods under $L(\cdot|\cdot)$.

126 **Comments for Dan.** Right now I'm not actually sampling from the neural network - instead, I
 127 enumerate the top few hundred lines of code suggested by the network, and then weight them by their
 128 likelihoods. So actually the form of NN is:

$$\text{NN}(p_L|p_1, \dots, p_{L-1}; x) \propto \begin{cases} 1, & \text{if } p_L \in \text{top hundred neural network proposals} \\ 0, & \text{otherwise.} \end{cases} \quad (7)$$

129 Do you think this is a problem? The neural network puts almost all of its mass on a few guesses. In
 130 order to get the correct line of code I sometimes need to get something like the 50th top guess, so I
 131 don't want to literally just sample from the distribution suggested by the neural network.

Algorithm 1 Neurally guided SMC

Input: Neural network NN, beam size N , maximum length L , target image x

Output: Samples of the program trace

Set $B_0 = \{\text{empty program}\}$

for $1 \leq l \leq L$ **do**

for $1 \leq n \leq N$ **do**

$p_n \sim \text{Uniform}(B_{l-1})$

$p'_n \sim \text{NN}(\text{render}(p), x)$

 Define $r_n = p'_n \cdot p_n$

 Set $\tilde{w}(r_n) = \frac{L(x|r_n)}{L(x|p_n)} \times \frac{\mathbb{P}[p'_n]}{\mathbb{P}[p'_n = \text{NN}(\text{render}(p), x)]}$

end for

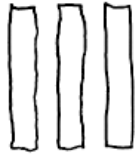
 Define $w(p) = \frac{\tilde{w}(p)}{\sum_{p'} \tilde{w}(p')}$

 Set B_l to be N samples from r_n distributed according to $w(\cdot)$

end for

return $\{p : p \in B_{l \leq L}, p \text{ is finished}\}$

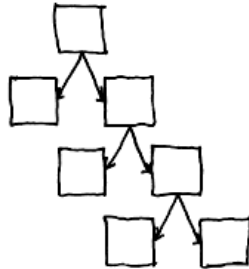
132 6 Preliminary Synthesis results



133

```
Rectangle(0,0,1,6)
Rectangle(2,0,3,6)
Rectangle(4,0,5,6)
```

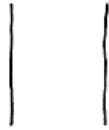
134



135

```
Rectangle(2,9,4,11)
for (3)
  Line(-2*i + 7,3*i + 3,-2*i + 6,3*i + 1,arrow = True,solid
  Line(2*i + 3,-3*i + 9,2*i + 4,-3*i + 7,arrow = True,solid
  Rectangle(2*i,-3*i + 6,2*i + 2,-3*i + 8)
  Rectangle(-2*i + 8,3*i,-2*i + 10,3*i + 2)
```

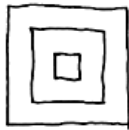
136



137

```
Line(0,0,0,5,arrow = False,solid = True)
```

138



139

```
Rectangle(0,0,5,5)  
Rectangle(1,1,4,4)  
Rectangle(2,2,3,3)
```

140

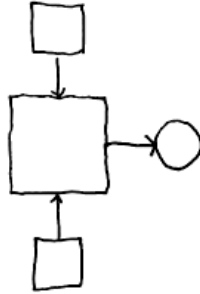


141

```
Circle(1,1)
```

142

143



144

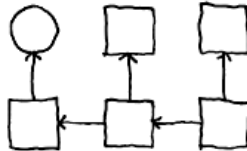
```
Rectangle(0,4,4,8)
  reflect(y = 12)
Circle(7,6)
Line(2,2,2,4,arrow = True,solid = True)
Line(4,6,6,6,arrow = True,solid = True)
Rectangle(1,0,3,2)
```

145



146

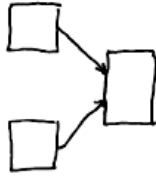
```
Rectangle(2,2,5,3)
Rectangle(0,0,3,1)
Rectangle(4,4,7,5)
```



147

```
Circle(1,5)
for (2)
  Line(-5*i + 9,-1*i + 2,-7*i + 9,-3*i + 4,arrow = True,solid = True)
  Rectangle(-4*i + 8,4*i,-4*i + 10,4*i + 2)
  reflect(x = 6)
  Line(7*i + 1,-1*i + 2,5*i + 1,-3*i + 4,arrow = True,solid = True)
  Rectangle(8*i,4*i,8*i + 2,4*i + 2)
```

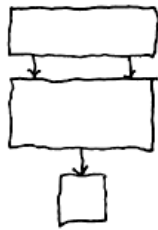
148



149

```
Rectangle(4,2,6,5)
    reflect(y = 7)
    Line(2,6,4,4,arrow = True,solid = True)
    Rectangle(0,5,2,7)
```

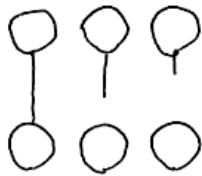
150



151

```
Line(3,3,3,2,arrow = True,solid = True)
Rectangle(0,7,6,9)
Rectangle(2,0,4,2)
Rectangle(0,3,6,6)
    Line(1,7,1,6,arrow = True,solid = True)
```

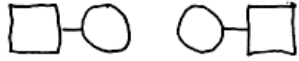
152



153

```
for (3)
    Circle(-3*i + 7,1)
    Circle(3*i + 1,6)
    Line(3*i + 1,2,3*i + 1,5,arrow = False,solid = True)
```

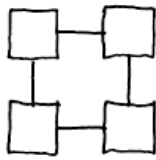
154



155

```
reflect(x = 12)
Circle(4,1)
Line(9,1,10,1,arrow = False,solid = True)
Rectangle(10,0,12,2)
```

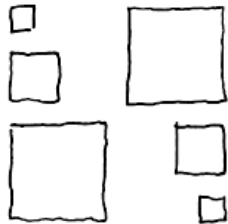
156



157

```
reflect(x = 6)
Line(5,2,5,4,arrow = False,solid = True)
    reflect(y = 6)
        Line(2,5,4,5,arrow = False,solid = True)
            Rectangle(0,4,2,6)
```

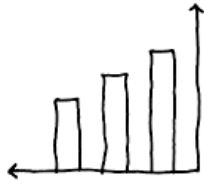
158



159

```
for (2)
    Rectangle(0,-3*i + 8,1,-2*i + 9)
    Rectangle(-3*i + 8,5*i,9,8*i + 1)
    Rectangle(-7*i + 7,-2*i + 2,-5*i + 9,4)
```

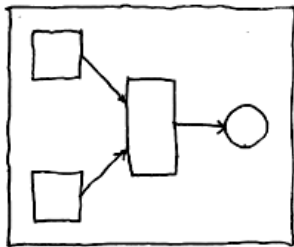
160



161

```
Rectangle(4,0,5,4)
for (2)
    Line(8,0,-8*i + 8,-7*i + 7,arrow = True,solid = True)
    Rectangle(-4*i + 6,0,-4*i + 7,-2*i + 5)
```

162



163

```
Circle(10,5)
Line(7,5,9,5,arrow = True,solid = True)
Rectangle(5,3,7,7)
Rectangle(0,0,12,10)
    reflect(y = 10)
    Line(3,2,5,4,arrow = True,solid = True)
    Rectangle(1,1,3,3)
```

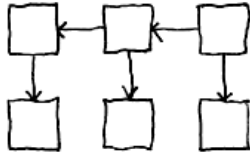
164



165

```
Line(0,0,0,2,arrow = False,solid = True)
Line(0,2,2,2,arrow = False,solid = True)
```

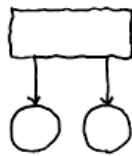
166



167

```
Line(8,5,6,5,arrow = True,solid = True)
Line(4,5,2,5,arrow = True,solid = True)
for (3)
    Line(-4*i + 9,4,-4*i + 9,2,arrow = True,solid = True)
    Rectangle(4*i,0,4*i + 2,2)
    Rectangle(-4*i + 8,4,-4*i + 10,6)
```

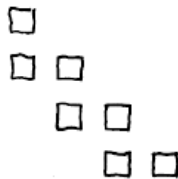
168



169

```
Rectangle(0,4,5,6)
reflect(x = 5)
Circle(4,1)
Line(1,4,1,2,arrow = True,solid = True)
```

170



171

```
Rectangle(0,6,1,7)
for (3)
    Rectangle(-2*i + 6,2*i,-2*i + 7,2*i + 1)
    Rectangle(-2*i + 4,2*i,-2*i + 5,2*i + 1)
```

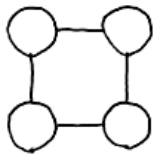
172



173

```
for (3)
  Circle(4*i + 1,1)
  Rectangle(4*i,0,4*i + 2,2)
```

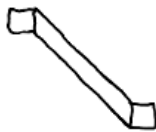
174



175

```
reflect(y = 6)
Line(2,1,4,1,arrow = False,solid = True)
  reflect(x = 6)
  Circle(1,1)
  Line(1,2,1,4,arrow = False,solid = True)
```

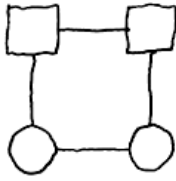
176



177

```
Line(1,5,5,1,arrow = False,solid = True)
Line(1,4,5,0,arrow = False,solid = True)
Rectangle(5,0,6,1)
Rectangle(0,4,1,5)
```

178



179

```
for (2)
  Circle(-5*i + 6,1)
  Line(-4*i + 6,-1*i + 2,-1*i + 6,-4*i + 5,arrow = False,solid
  Line(-1*i + 2,-4*i + 6,-4*i + 5,-1*i + 6,arrow = False,solid
  Rectangle(5*i,5,5*i + 2,7)
```

180



181

```
Line(0,0,0,5,arrow = False,solid = False)
Line(4,1,4,5,arrow = False,solid = False)
Line(4,0,4,1,arrow = False,solid = False)
```

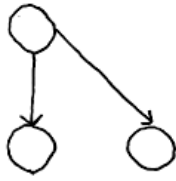
182



183

```
Rectangle(0,0,3,4)
```

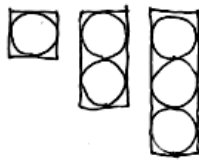
184



185

```
Circle(1,1)
for (2)
  Circle(-5*i + 6,5*i + 1)
  Line(1,5,5*i + 1,2,arrow = True,solid = True)
```

186



187

```
Circle(7,3)
for (3)
  Circle(-3*i + 7,5)
  Circle(-3*i + 7,2*i + 1)
  Rectangle(-3*i + 6,2*i,-3*i + 8,6)
```

188



189

```
Circle(1,8)
for (2)
  Line(4*i + 1,-5*i + 7,2*i + 3,5,arrow = False,solid = True)
  Rectangle(-4*i + 4,5*i,6,7*i + 2)
```

190



191

```
Circle(7,1)
Circle(4,1)
Circle(1,1)
```

192



193

```
Line(1,2,1,5,arrow = False,solid = True)
for (2)
  Line(2,-4*i + 4,-4*i + 6,4,arrow = False,solid = True)
  Line(0,-2*i + 6,-2*i + 2,6,arrow = False,solid = True)
  Line(1,5,2*i + 2,5,arrow = False,solid = True)
```

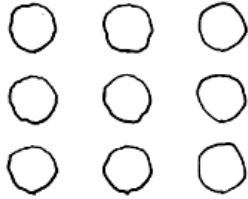
194



195

```
Rectangle(5,0,8,3)
Rectangle(2,1,4,3)
Rectangle(0,2,1,3)
```

196



197

```
for (3)
  Circle(-4*i + 9,4)
  Circle(-4*i + 9,1)
  Circle(4*i + 1,7)
```

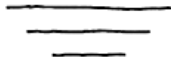
198



199

```
Circle(1,5)
Line(1,4,1,2,arrow = True,solid = True)
Rectangle(0,0,2,2)
```

200



201

```
Line(0,2,7,2,arrow = False,solid = True)
Line(1,1,6,1,arrow = False,solid = True)
Line(2,0,5,0,arrow = False,solid = True)
```

202

203



```

Circle(1,9)
Line(1,8,1,6,arrow = True,solid = True)
Line(1,3,1,2,arrow = True,solid = True)
Line(1,3,1,4,arrow = False,solid = True)
    reflect(y = 6)
    Circle(1,1)
  
```

204



205

```

Line(1,2,3,2,arrow = False,solid = True)
Line(2,1,4,1,arrow = False,solid = False)
Line(3,0,5,0,arrow = False,solid = True)
Line(0,3,2,3,arrow = False,solid = False)
  
```

206



207

```

Line(4,4,2,2,arrow = True,solid = True)
Rectangle(3,4,5,6)
Rectangle(0,0,2,2)
  
```

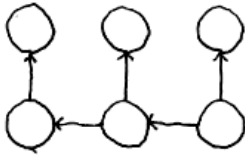
208

!

209

```
Line(0,0,0,4,arrow = False,solid = True)
```

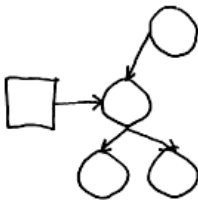
210



211

```
Line(4,1,2,1,arrow = False,solid = True)
for (2)
  Line(-4*i + 5,2,-4*i + 5,4,arrow = True,solid = True)
  for (3)
    Circle(-4*j + 9,4*i + 1)
    Line(8,1,3*i + 6,3*i + 1,arrow = True,solid = False)
```

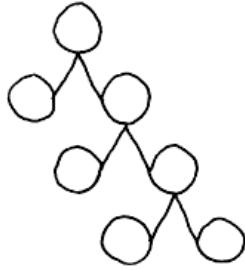
212



213

```
Rectangle(0,3,2,5)
for (2)
  Circle(3*i + 4,1)
  Circle(-2*i + 7,-3*i + 7)
  Line(5,3,-3*i + 7,2,arrow = True,solid = True)
  Line(4*i + 2,3*i + 4,4,4,arrow = True,solid = True)
```

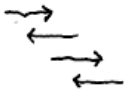
214



215

```
Circle(9,1)
for (3)
  Circle(2*i + 3,-3*i + 10)
  Circle(-2*i + 5,3*i + 1)
  Line(2*i + 2,-3*i + 7,2*i + 3,-3*i + 9,arrow = False,solid
  Line(-2*i + 7,3*i + 3,-2*i + 8,3*i + 1,arrow = False,solid
```

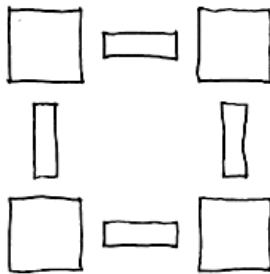
216



217

```
Line(5,0,3,0,arrow = True,solid = True)
Line(0,3,2,3,arrow = True,solid = True)
Line(3,2,1,2,arrow = True,solid = True)
Line(2,1,4,1,arrow = True,solid = True)
```

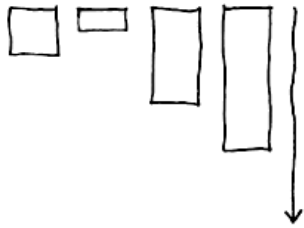
218



219

```
reflect(y = 11)
Rectangle(4,9,7,10)
  reflect(x = 11)
  Rectangle(8,0,11,3)
  Rectangle(1,4,2,7)
```

220



221

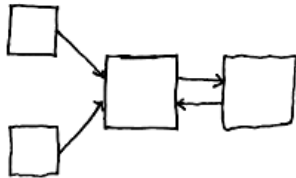
```
Line(12,9,12,0,arrow = True,solid = True)
for (2)
  Rectangle(-3*i + 6,3*i + 5,-3*i + 8,9)
  Rectangle(9*i,-4*i + 7,9*i + 2,9)
```

222

223

```
Line(4,0,0,0,arrow = False,solid = False)
```

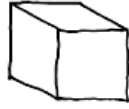
224



225

```
Line(8,3,9,3,arrow = False,solid = True)
for (2)
  Line(-6*i + 8,3*i + 3,-3*i + 7,3,arrow = True,solid = True)
  Line(5*i + 2,3*i + 1,5*i + 4,3,arrow = True,solid = True)
  Rectangle(9*i,2*i,10*i + 2,3*i + 2)
  Rectangle(-4*i + 4,3*i + 2,-5*i + 7,2*i + 5)
```

226



227

```
Line(0,1,0,4,arrow = False,solid = True)
Rectangle(2,0,5,3)
for (2)
    Line(0,3*i + 1,2,3*i,arrow = False,solid = True)
    Line(-3*i + 3,4,-2*i + 5,3,arrow = False,solid = True)
```

228 References

- 229 [1] SM Eslami, N Heess, and T Weber. Attend, infer, repeat: Fast scene understanding with generative models.
230 arxiv preprint arxiv:..., 2016. URL <http://arxiv.org/abs/1603.08575>.
- 231 [2] Daniel Ritchie, Anna Thomas, Pat Hanrahan, and Noah Goodman. Neurally-guided procedural models:
232 Amortized inference for procedural graphics programs using neural networks. In *Advances In Neural*
233 *Information Processing Systems*, pages 622–630, 2016.
- 234 [3] Brian Hempel and Ravi Chugh. Semi-automated svg programming via direct manipulation. In *Proceedings*
235 *of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 379–390, New
236 York, NY, USA, 2016. ACM.
- 237 [4] Haibin Huang, Evangelos Kalogerakis, Ersin Yumer, and Radomir Mech. Shape synthesis from sketches via
238 procedural models and convolutional networks. *IEEE transactions on visualization and computer graphics*,
239 2017.
- 240 [5] Gen Nishida, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Adrien Bousseau. Interactive
241 sketching of urban procedural models. *ACM Trans. Graph.*, 35(4), 2016.