EUROGRAPHICS 2018 / D. Gutierrez and A. Sheffer
(Guest Editors)

Volume 37 (2018), Number 2

Example-based Authoring of Procedural Modeling Programs with Structural and Continuous Variability

Daniel Ritchie[1] Sarah Jobalia[2] Anna Thomas[2]

[1]Brown University
[2]Stanford University

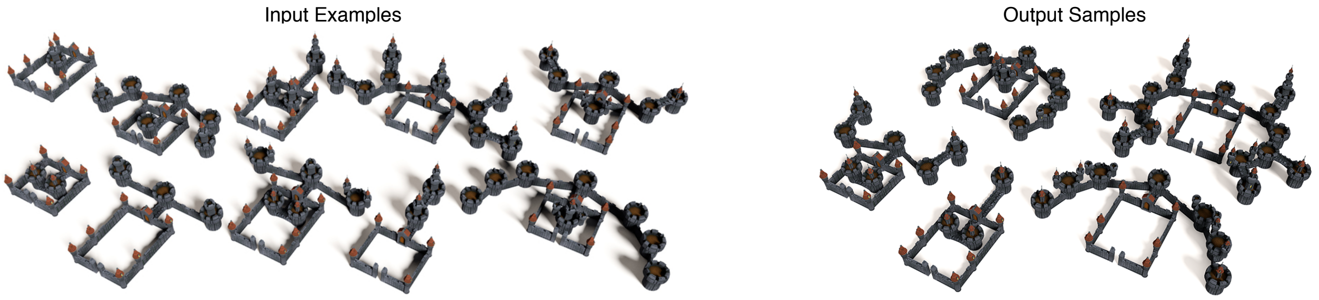

Figure 1: Our method takes part-based example objects as input *(Left)* and produces a program which generates more objects in the style of the examples *(Right)*. The program models both the hierarchical part structure and the continuous part transformations of the objects.

Abstract

Procedural models are a powerful tool for quickly creating a variety of computer graphics content. However, authoring them is challenging, requiring both programming and artistic expertise. In this paper, we present a method for learning procedural models from a small number of example objects. We focus on the modular design setting, where objects are constructed from a common library of parts. Our procedural representation is a probabilistic program that models both the discrete, hierarchical structure of the examples as well as the continuous variability in their spatial arrangements of parts. We develop an algorithm for learning such programs from examples, using combinatorial search over program structures and variational inference to estimate continuous program parameters. We evaluate our method by demonstrating its ability to learn programs from examples of ornamental designs, spaceships, space stations, and castles. Experiments suggest that our learned programs can reliably generate a variety of new objects that are perceptually indistinguishable from hand-crafted examples.

CCS Concepts
•***Computing methodologies*** $\rightarrow$ *Probabilistic reasoning; Neural networks; Shape analysis;*

1. Introduction

Procedural modeling is a powerful technique for creating varied, detailed visual content with minimal human effort. It has long been used to generate trees, buildings, cities, and decorative patterns for games, animation, and other applications [PJM, MWH*, WZS]. Procedural modeling is especially well-suited for creating content by combining pre-made assets. Such 'modular design' is popular in game production to increase content variety without significant artist effort: as of time of writing, there were over 1700 results for the query 'modular' on the Unity Asset Store [Uni].

However, authoring procedural models is difficult, requiring both programming expertise and an artistic eye—few people possess enough of both to be up to the task. To eliminate the programming burden, an attractive idea is to learn a procedural representation from a small number of example models. In the graphics literature, grammars are the most popular procedural representation, and prior work has addressed the problem of inducing probabilistic grammars from modular design examples [TYK*]. However, these grammars capture only the discrete, hierarchical structure of the examples—the spatial relationships, or arrangements, of parts are not considered. One could instead use newer methods based on adversarial autoencoders to learn a generative model of both hierarchical part structure and arrangement [LXC*]. But these methods require significant training data, making them unsuitable for the setting where a single user provides a few examples.

In this paper, we present a method for learning a generative, procedural model of both part structure and arrangement from a small number of modular design examples. Our system takes as input a library of modular parts and a small number of connectivity hierarchies constructed from those parts (no more than ten, in our experiments). It then learns a probabilistic program that generates the examples with high probability, as well as more objects like them. These programs resemble probabilistic grammars, but they allow for more structural variability, and they capture continuous variability by using simple neural networks to model distributions over rigid part transformations. Like previous grammar induction methods, our approach finds programs by a combinatorial search over possible program structures. For each candidate program, it must evaluate the likelihood of the examples under that program. Since our programs contain neural network components, this is made possible by recent advances in black-box variational inference [RGB].

We demonstrate the capability of our method by learning programs that generate ornamental designs, spaceships, space stations, and castles. We also experimentally evaluate the generalization capability of the learned programs, as well as conduct a perceptual study in which objects generated by our learned programs were judged to be indistinguishable from hand-created examples.

Our contributions are:

- A probabilistic program representation that models both the hierarchical structure and the continuous arrangement of modular part-based objects.
- An algorithm for learning this representation from a small number of example objects.
- Qualitative and quantitative evaluations of the proposed method's ability to reliably capture and generalize the examples.

After reviewing related work in Section 2, Section 3 gives a high-level overview of our approach. We then describe in more detail how our method pre-processes examples (Section 4), the program representation we use (Section 5), and how we learn such programs from the examples (Section 6). Finally, Section 7 presents qualitative and quantitative evaluations of our method.

2. Related Work

Learning Grammars and Programs The computer graphics and vision literatures feature several applications of grammar learning for analyzing and synthesizing visual content. Most related to our method are the probabilistic grammar induction systems of Talton et al. and Martinovic et al. [TYK*, MVG]. The method of Talton et al. learns grammars describing discrete structures, discarding continuous part transformations and representing examples as labeled trees. The method of Martinovic et al. learns 2D split grammars with the help of an algorithm for parsing irregular 2D lattices. Our method uses a similar combinatorial search procedure as these approaches, but its program representation is more flexible than a grammar and can model continuous object part transformations. Other prior work uses probabilistic grammars to parse over-segmented 3D scene graphs [LCK*]. This method considers object spatial relationships, but it is designed for analyzing existing scene graphs and makes independence assumptions about both structure and spatial arrangement that are not appropriate for synthesizing new scenes. The Probabilistic Scene Grammars framework uses a grammar-based probabilistic graphical model to localize faces and contours in images [CF16]. This system also models spatial relationships but relies on hand-crafted grammars and is defined on discretely-sampled pixel grids. There also exist techniques for inferring a non-probabilistic grammar from a single mesh or point cloud, vector artwork, or building facade [BWS, SBM*, WYD*]. In contrast, we focus on probabilistic models to capture the variation in a set of examples. Other work has explored refining the structure of an existing grammar through user-provided preference ratings of its outputs [DLC*]; our work focuses on learning new programs from examples.

Researchers in cognitive science have learned probabilistic programs to model visual concepts. Ellis et al. use SMT solvers to learn programs that generate coherent collections of abstract shapes, and Lake et al. use a custom inference procedure to learn programs that generate handwritten characters [ESLT, LST15]. Most relevant to our method is the work of Huang et al., which learns recursive probabilistic programs which generate tree structures [HSG11]. Like the grammar induction system of Talton et al., however, it does not model continuous spatial transformations.

Part-Based Shape Synthesis Outside of grammar/program induction-based methods, there exist other approaches for synthesizing part-based shapes. Some of these systems take a starting shape and swap its parts with other parts, where swap compatibility is determined by connectivity and symmetry analysis [JTRS, LVW*]. More related to our method are approaches that learn generative models of component-based shapes, either for suggesting parts to add to a shape or for creating new shapes [CKGK, KCKK]. These systems use Bayesian Networks to model non-hierarchical, non-recursive part assemblies; similar methods have also been used to model the attributes of residential building exteriors [FW16]. Our method uses probabilistic programs to model (potentially recursive) hierarchical structure. The recent GRASS system learns generative models of shape hierarchies using recursive autoencoders [LXC*]. GRASS represents shapes as points on a latent manifold, which requires large shape collections to reliably learn. In contrast, our work focuses on learning from a small set of examples created by a single user.

Inverse Procedural Modeling There is a long line of research on finding executions of a known procedural model that satisfy some criteria, such as matching a desired output [TLL*11, SPK*14, VGDA*, RMGH, RTHG, NGDA*, HKYM16]. In contrast, our work seeks to infer an unknown procedural model from multiple example outputs. These two problems have both been referred to as 'inverse procedural modeling'; our work focuses only on the latter.

Black-Box Variational Inference Variational inference (VI) methods approximate intractable probability distributions by optimizing the parameters of simpler, tractable distributions. Recent advances in VI have enabled efficient learning of complex latent variable models, including those that include neural net-

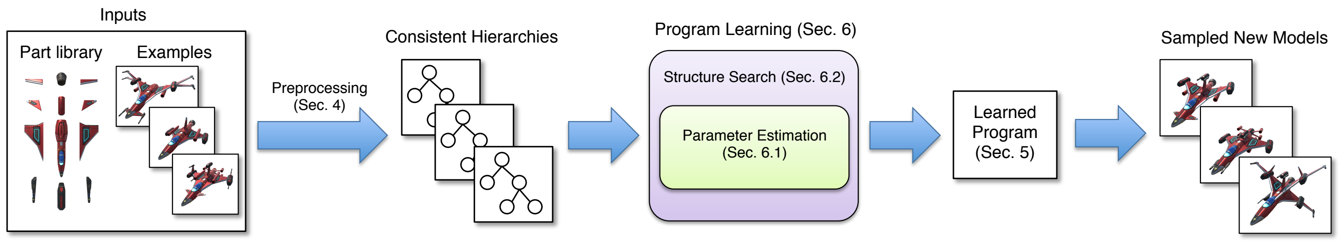

Figure 2: Pipeline of our program learning approach. The system takes as input several example part connectivity hierarchies assembled from a common part library. It preprocesses these examples into a consistent form, and then feeds them to an iterative program structure search algorithm. The returned program generates new objects in the style of the examples.

work components, via stochastic gradient-based parameter optimization. General-purpose VI techniques have been developed for discrete latent variables [RGB, MG, GLSM], continuous latent variables [KW], and combinations of the two [RHG16, SHWA]. Our method uses VI to fit a program to a set of example hierarchies, where the choices of which subroutines generate which sub-trees of the examples are treated as discrete latent variables.

3. Approach

Many modular 3D objects decompose into a tree-structured connectivity hierarchy of parts. Similarly, when a program runs, it executes a tree of function calls: one function calls several other functions, which themselves call other functions, and so on. We make the assumption that a program which generates a tree-structured object does so via a function call tree which is isomorphic to the object hierarchy. Thus, a tree-structured example object can be viewed as an *execution trace* from the program to be learned.

Figure 2 shows an overview of our program learning approach. Our system takes as input a small set of connectivity hierarchies assembled from a common library of 3D parts. It first preprocesses these examples to transform them into a consistent form (Section 4). As described above, the system treats these consistent hierarchies as execution traces from our learnable program representation (Section 5). Program learning is phrased as a search problem: the system searches for the simplest probabilistic program which assigns high probability to the examples (formalized in Section 6). The outer loop of this process is a combinatorial search over possible program structures and resembles prior work on probabilistic grammar and program induction [TYK*, MVG, HSG11]. At each search iteration, an inner loop optimizes the parameters of the current structure to maximize the probability of the examples. When executed, the program returned by the search process generates new, random models in the style of the examples.

4. Example Preprocessing

The input to our system is a set of connectivity hierarchies assembled from a common library of 3D parts. Each part in the library is labeled with a type $\mathcal{T}$, such that all parts with the same type have the same geometry and local coordinate frame. We make no assumptions about these local coordinate frames, e.g. the origin does not need to correspond to the part's center of mass. An input connectivity hierarchy can be any hierarchical scene graph assembled from these parts, where each node corresponds to one part. Figure 3a-b shows some example input models and their logical part hierarchies. Each hierarchy node stores its part type as well as its affine transformation (not shown, for figure clarity).

Imagine playing the role of a program which generates models like these. When generating the *Body* node, what configurations of its child nodes should you allow? You should only generate *Wings* and *Weapons* in reflectionally-symmetric pairs, since e.g. a ship with an odd number of wings would look implausible. You also should not generate certain child parts simultaneously: the *Engine* and *Thruster Cap* would be implausibly colocated if both occurred in the same model. Howevever, other child parts can safely co-occur, such as the *Engine* and *Weapons*. In this section, we describe how we preprocess the input examples to explicitly encode this information for the learnable program to exploit.

First, we encode information about symmetries. For each node in the input example hierarchies, we extract all translational, rotational, and reflectional symmetries that exist among its children; these may be nested (e.g. a translational symmetry group of reflectional symmetry groups). We compute symmetry groups following prior work on symmetry hierarchy construction [WXL*]. Figure 3c shows the input hierarchies after symmetry group extraction.

Next, we encode information about co-occurrence dependencies between child nodes. Probabilistic grammars assume that *all* children of a nonterminal node are dependent: each grammar production rule is a joint distribution over possible lists of children. This is guaranteed not to generate any implausible child configurations, but it also cannot generalize beyond the exact configurations that occur in the examples. One could go to the opposite extreme and treat all children as independent, as in prior work on parsing over-segemented scene graphs [LCK*]. This approach generalizes to new configurations but may miss key dependencies (e.g. the *Engine*/*Thruster Cap* conflict mentioned above). We would like our programs to fall somewhere in between: to generalize beyond example child configurations while also respecting important co-occurrence dependencies. In what follows, we describe a simple scheme for identifying such dependencies and encoding them in the input hierarchies so that learned programs respect them.

For each part type $\mathcal{T}$ (e.g. *Body*), we first build a set of all possible children that nodes of type $\mathcal{T}$ could have. We call each entry in this set $\mathcal{S}^{\mathcal{T}}$ a *slot*: let $\mathcal{S}^{\mathcal{T}} = \{s_1^{\mathcal{T}}, s_2^{\mathcal{T}}, \ldots\}$, where $s_i^{\mathcal{T}}$ is the ith slot of parent part type $\mathcal{T}$. If a type of child node occurs multiple times in the examples, there is one slot for each possible occurrence, up

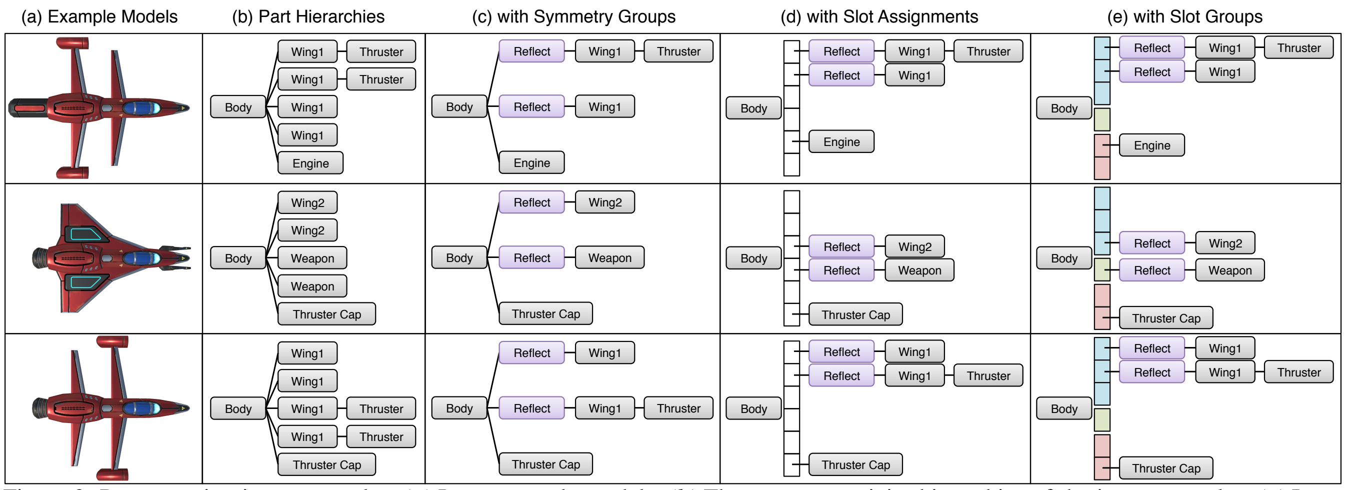

Figure 3: Preprocessing input examples. *(a)* Input example models. *(b)* The part connectivity hierarchies of the input examples. *(c)* Input hierarchies after extracting symmetry groups. *(d)* Input hierarchies after determining possible child slots and assigning child nodes to those slots. *(e)* Input hierarchies after grouping slots based on the overlap of their assigned nodes; each slot group has a unique color.

to the maximum number of times that child occurrs (e.g. there can be two *Reflect-Wing1* nodes in the examples in Figure 3). Thus, the total number of slots for type $\mathcal{T}$ is the sum of maximum number of each of its possible child node types (e.g. for *Body*, there are 2 *Reflect-Wing1* slots + 1 *Reflect-Wing2* slot + 1 *Reflect-Weapon* slot + 1 *Engine* slot + 1 *Thruster Cap* slot = 6 slots total). We then assign each child node in the examples to a slot. For child node types which occur only once (e.g. *Engine*), those child nodes are assigned to their one available slot. For child node types which can occur multiple times (e.g. *Reflect-Wing1*), we must decide to which of the multiple slots available each node belongs. We would like to create slot assignments such that spatially 'corresponding' nodes are placed in the same slot. For example, in Figure 3d, the two rear-most occurrences of *Reflect-Wing1* are placed in the same slot (the first slot), as are the two forward-most occurrences (the second slot).

We next formalize this notion of correspondence and describe how we compute maximally-corresponding slot assignments. Let $\mathcal{N}(s_i^{\mathcal{T}})$ denote the set of example nodes assigned to slot $s_i^{\mathcal{T}}$. We wish to find the slot assignment $\mathcal{N}$ which minimizes the following objective:

$$\min_{\mathcal{N}} f(\mathcal{N})$$
$$f(\mathcal{N}) = \max_{s_i^{\mathcal{T}} \in \mathcal{S}^{\mathcal{T}}} \max_{\mathbf{n}_i, \mathbf{n}_j \in \mathcal{N}(s_i^{\mathcal{T}})} d(\mathbf{n}_i, \mathbf{n}_j) \quad (1)$$

where $d(\mathbf{n}_1, \mathbf{n}_2)$ is the Hausdorff distance between the geometries of nodes $\mathbf{n}_1$ and $\mathbf{n}_2$ in the coordinate frame of the parent part $\mathcal{T}$. Since all child nodes assigned to a slot have the same type $\mathcal{T}_{\mathbf{child}}$ and thus the same geometry, two nodes $\mathbf{n}_1$ and $\mathbf{n}_2$ which are colocated in their parent coordinate frame have $d(\mathbf{n}_1, \mathbf{n}_2) = 0$. The objective f penalizes any large distance between any two nodes assigned to the same slot, reflecting the intution that children which occur in a similar configuration with respect to their parent should be placed in the same slot.

For each possible parent node type $\mathcal{T}$ and each of its possible child node types $\mathcal{T}_{\mathbf{child}}$, we solve this optimization problem greedily over all the input examples by adding one example node of type $\mathcal{T}_{\mathbf{child}}$ at a time to the slot assignment $\mathcal{N}$. To initialize, we choose a node of type $\mathcal{T}$ from the examples that has the maximum possible number of $\mathcal{T}_{\mathbf{child}}$-type children and arbitrarily assign those children to slots. Every slot of type $\mathcal{T}_{\mathbf{child}}$ in $\mathcal{N}$ now has one node assigned to it. We then repeat the following until all $\mathcal{T}$-type nodes in the examples have had all their $\mathcal{T}_{\mathbf{child}}$-type children assigned to slots: for each remaining $\mathcal{T}$-type node in the examples, we try all possible assignments of its $\mathcal{T}_{\mathbf{child}}$-type children to slots and compute Equation 1 for the resulting overall slot assignment. We select the node that admits the lowest-cost overall assignment and set $\mathcal{N}$ to that assignment.

With part children now in this slot format, finding co-occurence dependencies can be phrased as finding any slots whose assigned nodes overlap. We call a set of slots connected via overlap relations a *slot group*. We compute the slot groups for part type $\mathcal{T}$ by checking for intersections between the nodes assigned to each pair of slot groups $\mathcal{N}(s_i^{\mathcal{T}})$, $\mathcal{N}(s_j^{\mathcal{T}})$. To support the common modular design idiom where parts interpenetrate their parent part, we compute the boolean difference of parts from their parents before checking for intersections (to avoid detecting spurious, invisible collisions that occur inside the parent part). Figure 3e shows slot groups, where slots in the same group have the same color.

We emphasize that a slot group does *not* represent a hard constraint that only one of its slots may occur at a time, or that intersecting slots cannot co-occur. Rather, a slot group signals that there may be some dependency between which slots are allowed to co-occur. It is the responsibility of the program representation described in the next section to *learn* which slots in the same slot group can co-occur. For instance, we would expect the program to learn that for the blue slot group in Figure 3e, either the first two slots occur or just the third slot occurs (corresponding to a ship with either two thin pairs of wings or one wide pair of wings). In the extreme case where all slots overlap and are placed into a single slot group, dependencies exist between all slots, which restricts the program to generating only child part combinations that are observed in the examples—the same behavior as a grammar.

We also note that imposing dependencies between overlapping child parts is the minimum dependency that must be modeled to avoid generating visually implausible results. The example author may intend for other *functional* dependencies to be respected, as well—for example, she may intend that ships only have a particular type of wing when some type of engine is present. Such extra dependencies could be incorporated by allowing the author to annotate parts as dependent; our system could then force such parts into the same slot group.

5. Learnable Program Representation

In this section, we describe a program representation that is learnable from the processed example hierarchies described in the previous section. As stated in Section 3, we assume that the structure of this program's execution mirrors the hierarchical structure of the examples. The program must also model the continuous transformations of each hierarchy node, as well as any symmetries.

Algorithm 1 shows pseudocode for our learnable program representation. The program assumes that there are multiple subroutines, or *abstractions*, available for it to call. For instance, a spaceship-generating program might have one abstraction to generate the body node and another to generate wing nodes. Abstractions are identified by a unique ID *absID*. They are also typed: an abstraction either generates an object part node or one of the three types of symmetry nodes (i.e. reflection, translation, rotation). The program dispatches on this type to call an appropriate generating function for that type of abstraction. Algorithm 1 shows two of these functions, one for part node abstractions (GENPARTNODE) and one for reflectional symmetry abstractions (GENREFLSYMNODE). In this code, GENNODE is the main function which dispatches to the appropriate generator function based on the type of *absID*. As the program is probabilistic, these functions sample from probability distributions (e.g. Discrete, Gaussian). Distributions are parameterized by parameters θ, which are indexed by program state variables (the current abstraction, slot, and slot group) as appropriate.

The program dispatches to GENPARTNODE to generate object part nodes. It first samples a part from a discrete distribution over possible parts. To generate the node's children, it proceeds in two steps. First, for each slot group of the selected part, it samples a Bernoulli random variable indicating whether each slot in the group should be used. To model dependencies between slots, these variables are sampled jointly from a neural autoregressive distribution estimator, or NADE [UCG*16]. A NADE defines a joint distribution over multiple random variables by factorizing it into a product of conditional distributions using the chain rule for probabilities: the second variable is conditional on the first, the third is conditional on the first and second, etc. The i^{th} conditional is implemented as a feedforward neural network $f_i(\mathbf{x}_1, \mathbf{x}_2 \ldots \mathbf{x}_{i-1})$ that takes the values of the previous $i-1$ variables as input and outputs the distribution parameters for the i^{th} variable (in this case, a single scalar Bernoulli probability p). This formulation allows the

Algorithm 1 Pseudocode for our learnable program representation

```
// Generate an object part node
procedure GENPARTNODE(absID, xform)
    partID ~ Discrete($\theta_{\mathbf{part}}$ (absID))
    children ← [ ] , i ← 1
    for slotGroup ∈ slotGroups[partID] do
        usedSlots ~ BernoulliNADE($\theta_{\mathbf{slots}}$ (absID, slotGroup))
        for slot ∈ slotGroup | usedSlots[slot] = true do
            absID′ ~ Discrete($\theta_{\mathbf{abs}}$ (absID, slot))
            xform′ ~ TransformDist($\theta_{\mathbf{xform}}$ (absID, slot))
            children[i] ← GENNODE(absID′, xform′)
            i ← i + 1
    return MAKEPARTNODE(partID, xform, children)
// Generate a reflectional symmmetry node
// (Logic for other types of symmetries is similar)
procedure GENREFLSYMNODE(absID)
    axis ~ Gaussian($\theta_{\mathbf{reflAxis}}$ (absID))
    absID′ ~ Discrete($\theta_{\mathbf{abs}}$ (absID, 1))
    xform′ ~ TransformDist($\theta_{\mathbf{xform}}$ (absID, 1))
    child ← GENNODE(absID′, xform′)
    return MAKEREFLSYMNODE(axis, child)
```

NADE to model complex, potentially nonlinear relationships between variables. For each used slot, GENPARTNODE samples an abstraction to call as well as a transform to apply to the generated child node. We describe the transform distribution at the end of this section.

GENREFLSYMNODE is executed to generate reflectional symmetry nodes. Such nodes have only one child (i.e. the reflectionally-symmetric sub-component) for which an abstraction and transform must be sampled. GENREFLSYMNODE also samples a reflection axis (i.e. the reflection plane normal) from a Gaussian distribution. Our implementation samples the axis in spherical coordinates, drawing the polar and azimuthal angles independently. The generators for translational and rotational symmetry nodes (not shown) sample their symmetry-related quantities, such as rotation axis and center of rotation, from independent Gaussians in a similar fashion.

Transform distributions We represent the affine transformation of a node as three axis-aligned scales, rotations, and translations. These quantities can be strongly correlated, and the correlations may be nonlinear: for example, the rotation of a wing of a spaceship in Figure 2 determines its translation. Thus, we model the distribution over transformations using a Gaussian NADE (i.e. a single variable-variance Gaussian RNADE [UCG*16]). Since scales must be positive, we model the distribution over log-scales. To account for negative scales, we use Bernoulli variables to model the sign of each scale component. Each translation component is represented as a percentage of the parent part's bounding box length along that dimension; this makes the model more scale-invariant.

NADE architecture details The Bernoulli NADE that models slot usage (Algorithm 1, line 6) uses a hidden layer of the same size as its output layer (i.e. the number of slots in the slot group) with a sigmoid activation. The Bernoulli NADE that models the sign of each transform scale component also has a hidden layer of the same size as its output layer (i.e. 3) with a sigmoid activation. The Gaussian NADE that models continuous transform components also uses a hidden layer of the same size as its output layer (i.e. 9, 3 for each of log-scale, rotation, and translation). It uses a tanh activation for its hidden layer. Since a NADE factorizes a joint distribution into a sequence of conditionals, there is a free choice of how to order those conditionals. Our Gaussian NADE predicts log-scales, then rotations, then translations, following the order used by many 3D modeling packages. Within each of those categories, it predicts individual components in decreasing order of variance as observed in its training data. This ordering helps prevent overfitting, since we are learning these NADEs from very small data. Predicting the components with the most variation first prevents the network from learning to predict that variance based on spurious minor variations in other, lower-variance components.

6. Learning Programs from Examples

The program described in the previous section calls one of multiple abstractions at each step of its execution. However, we are not told in advance how many abstractions the program should have. Furthermore, for each abstraction, we do not know what its parameters θ should be. This information must be learned from the examples. In this section, we describe our approach to program learning.

Formally, we wish to learn a program $\mathcal{P} = (\mathcal{A}, \theta)$. $\mathcal{A} = \{\mathbf{a}_i\}$ is the set of abstractions available to the program, and $\theta(\mathbf{a})$ are the parameters associated with abstraction $\mathbf{a}$. Let $\mathcal{T}_\mathbf{a}$ be the type of node generated by abstraction $\mathbf{a}$, and let $\mathcal{A}(\mathbf{a}, s_i^{\mathcal{T}_\mathbf{a}})$ be the set of abstractions that can be called from slot $s_i^{\mathcal{T}_\mathbf{a}}$ of abstraction $\mathbf{a}$, The goal of program learning is to find a program that generates the examples with high probability but also generalizes from them to generate other, similar objects. As in prior work, we follow the minimum description length principle: the best explanation for a set of data is the most compact one [Gru07]. Thus, given a dataset of examples $\mathbf{D}$, we prefer programs that maximize the following posterior log probability objective:

$$\log p(\mathcal{P} \mid \mathbf{D}) \propto \log p(\mathbf{D} \mid \mathcal{P}) - \lambda \cdot C(\mathcal{P}) = O_\mathbf{D}(\mathcal{P}) \qquad (2)$$

$C(\mathcal{P})$ is a measure of the complexity of the program $\mathcal{P}$:

$$C(\mathcal{P}) = |\mathcal{A}| + \sum_{\mathbf{a} \in \mathcal{A}} \left[\sum_{s_i^{\mathcal{T}_\mathbf{a}} \in \mathcal{S}^{\mathcal{T}_\mathbf{a}}} |\mathcal{A}(\mathbf{a}, s_i^{\mathcal{T}_\mathbf{a}})| \right]$$

$C(\mathcal{P})$ counts the number of abstractions used by the program as well as the number of abstractions which can be called from each available abstraction slot. λ controls the weight of this complexity prior; we use $\lambda = 1$ for all results reported in this paper.

Given the above definitions, program learning proceeds as follows. We initialize with a program $\mathcal{P}_0$ which can generate the examples. A *structure search* process then repeatedly generates modifications $\mathcal{P}'$ to the abstractions available to the current program $\mathcal{P}$. We optimize the parameters θ of $\mathcal{P}'$ to maximize the data log likelihood $\log p(\mathbf{D} \mid \mathcal{P}')$. A modification is accepted if $O_\mathbf{D}(\mathcal{P}') > O_\mathbf{D}(\mathcal{P})$. In the next section (6.1), we describe how we optimize program parameters to maximize the data log likelihood. Then, in the following section (6.2), we describe the program structure search procedure.

6.1. Optimizing Program Parameters

To determine how well a program represents the examples, we must first optimize the program's parameters to maximize the probability of it generating the examples.

Consider the random variables used by the program in Algorithm 1. The variables for part IDs, slot usages, transforms, and symmetries are all *observed* variables: their values are given in the input example hierarchies. However, the discrete choice of which abstraction generates each hierarchy node is not known—these are *latent* variables. Furthermore, some variables are drawn from distributions parameterized by neural networks. Thus, we require a parameter estimation method that works in the presence of both discrete latent variables and neural network model components.

If $\mathbf{y}$ denotes the values of all observed variables in the example, and $\mathbf{x}$ denotes a possible assignment of values to all latent variables, then the data log likelihood can be written as:

$$\mathcal{L}_\mathbf{D}(\mathcal{P}) = \log p(\mathbf{D} \mid \mathcal{P}) = \sum_\mathbf{x} \log p(\mathbf{x}, \mathbf{y} \mid \mathcal{P})$$

This objective is intractable in general, as there are an exponential number of possible latent variable assignments $\mathbf{x}$. Instead, a tractable alternative is the mean-field variational lower bound on the data log likelihood [RGB]:

$$\tilde{\mathcal{L}}_\mathbf{D}(\mathcal{P}) = \mathbb{E}_{\mathbf{x} \sim q(\cdot)} \left[\log p(\mathbf{x}, \mathbf{y} \mid \mathcal{P}) - \sum_{x_i \in \mathbf{x}} \log q_i(x_i) \right] \qquad (3)$$

Instead of enumerating all possible latent variable assignments $\mathbf{x}$, we sample each latent abstraction choice variable x_i from an auxiliary distribution q_i. Each q_i is an independent, discrete distribution over the possible abstractions for variable x_i whose parameters are optimized jointly with the program parameters θ. In effect, each q_i learns which abstractions are most likely for x_i.

We estimate the gradient of $\tilde{\mathcal{L}}_\mathbf{D}(\mathcal{P})$ using the score function estimator [SHWA]. This estimator is simple to implement and supports discrete latent variables, but it is known to have high variance, which can lead to unstable gradients. Our prototype program learning system is written in the WebPPL probabilistic programming language [GS14], which provides several established variance-reduction techniques [RHG16].

We use the Adam stochastic gradient optimizer [KB] to maximize Equation 3. During optimization, we linearly decrease the temperature of the softmax function used to normalize each q_i distribution. This pushes the q_i's toward deterministic functions which can be interpreted as hard assignments of abstractions to example hierarchy nodes. When the q_i's become deterministic, the lower bound $\tilde{\mathcal{L}}_\mathbf{D}(\mathcal{P})$ converges to the data likelihood $\mathcal{L}_\mathbf{D}(\mathcal{P})$.

6.2. Program Structure Search

The parameter optimization problem posed in the previous section is challenging to solve, as it requires finding optimal assignments of abstractions to all nodes in the input example hierarchies. If we simply posit that there exist N abstractions of each type and try to optimize Equation 3 directly, the optimizer easily becomes stuck in one of many spurious local optima. We need a principled way to incrementally explore the space of programs, rather than trying to solve for the best program directly in one shot.

To this end, we turn to ideas from probabilistic grammar induction. One approach to inducing probabilistic grammars from positive examples is Bayesian model merging: start with an over-specialized grammar that generates all of and only the examples, then repeatedly merge grammar non-terminals to construct progressively more general grammars [SO94, TYK*, MVG]. Another approach starts with a small, over-generalizing grammar and repeatedly splits non-terminals [HM98]. Either approach is applicable in our setting, since program abstractions are analagous to grammar non-terminals. We experimented with both and chose the latter, as we found that search starting from a program with a small number of abstractions could find a compact, generalizing program in fewer iterations than merge-based search starting with many abstractions. Minimizing the number of search iterations is always desirable but is especially important in our method: as we discuss below, each iteration requires a non-trivial stochastic gradient optimization.

Initialization We initialize search with a program $\mathcal{P}_0$ that has one abstraction for each object part type. For each slot, $\mathcal{P}_0$ also has one symmetry node abstraction for each type of symmetry that occurs in that slot. We run parameter estimation for 500 iterations to set the parameters of $\mathcal{P}_0$. Early experiments that initialized with fewer abstractions (e.g. only one part node abstraction) took more iterations to find good results, did so less reliably, and passed through states that resemble our chosen initialization state.

Abstraction Splitting Given a current program $\mathcal{P}$, search repeatedly splits its abstractions. After initialization, all abstractions $\mathbf{a}$ in $\mathcal{P}$ are inserted into a priority queue ordered by the number of times $\mathbf{a}$ is used by $\mathcal{P}$ to generate example hierarchy nodes. This heuristic reflects the intuition that frequently-used abstractions are more likely to benefit from division into two more specialized ones. Search then repeatedly picks the first abstraction $\mathbf{a}$ out of this queue and splits it, where splitting consists of:

1. Replace $\mathbf{a}$ with two duplicates of itself, $\mathbf{a}'_1$ and $\mathbf{a}'_2$.
2. For any q_i distributions whose domain includes $\mathbf{a}$, divide the probability assigned to $\mathbf{a}$ evenly between $\mathbf{a}'_1$ and $\mathbf{a}'_2$.
3. Re-optimize the program parameters by running parameter optimization for 500 iterations. After re-optimization, q_i's involving $\mathbf{a}'_1$ and $\mathbf{a}'_2$ will have converged to assigning all probability to only one of the two new abstractions.
4. For all discrete distributions over abstraction choices (lines 8 and 17 in Algorithm 1), snap all parameters smaller than a threshold $\tau_{\mathbf{snap}} = 0.02$ to zero. This allows updating the $\mathcal{A}(\mathbf{a}, s_i^{\mathcal{T}(\mathbf{a})})$'s for each slot $s_i^{\mathcal{T}(\mathbf{a})}$) to remove abstractions that now have zero probability of being generated at that slot.

If the score is higher for this post-split program $\mathcal{P}'$ ($O_{\mathbf{D}}(\mathcal{P}') > O_{\mathbf{D}}(\mathcal{P})$), the search process accepts this split ($\mathcal{P} \leftarrow \mathcal{P}'$). On acceptance, $\mathbf{a}'_1$ and $\mathbf{a}'_2$ are inserted into the queue, as they may require further splitting. In addition, any previously-rejected abstraction $\tilde{\mathbf{a}}$ which can call $\mathbf{a}'_1$ or $\mathbf{a}'_2$ is also re-inserted into the queue, as it might now split successfully. Search terminates when the queue is empty, i.e. search has attempted to split every abstraction.

Two-Phase Search To improve both the speed and reliability of the above search procedure, we divide it into two phases. In the first phase, we remove all continuous variables from the program (i.e. part transformations and symmetry parameters) and run search to termination. This phase focuses on finding an appropriately-generalized representation of the hierarchical structure of the examples, but further splits may still be needed to accurately model continuous spatial relationships. This division speeds up search, since the program runs faster with continuous distributions excluded. It also makes search more reliable, since we have found that including continuous variables early on in the search procedure tends to produce too many competing concerns in the parameter optimization objective. This leads to high-variance gradients and q_i's that converge to poor local optima.

In the second phase, we add the continuous variables back to the program. Some of these variables are deterministic: for instance, a part that has the same orientation in every input example. If search optimizes both the means μ and standard deviations σ of the Gaussian distributions for such variables, the σ's are driven to zero, producing arbitrarily-large and unstably-fluctuating values of the data log-likelihood $\mathcal{L}_{\mathbf{D}}(\mathcal{P})$. Such values of $\mathcal{L}_{\mathbf{D}}(\mathcal{P})$ cannot reliably be used to make split accept/reject decisions. Thus, for the duration of this phase, we fix the σ's to constant values which reflect conservative estimates about how much continuous variation we expect a single abstraction to exhibit. We use one parameter for angular variables ($\sigma_{\mathbf{rot}} = \pi/8$), one for log-scale variables ($\sigma_{\mathbf{scale}} = \log(1.1)$), and one for translational variables ($\sigma_{\mathbf{trans}} = 0.2$). From a Bayesian perspective, this design can be interpreted as placing a Dirac delta prior on the σ parameters (e.g. for angular variables, $p(\sigma) = \delta(\sigma - \sigma_{\mathbf{rot}})$. Smooth priors are also possible, though we have not experimented with them. Note that modeling transform distributions with NADEs facilitates this process by providing a single controllable variance parameter for each transform component (as opposed to e.g. multivariate Gaussians, which additionally use multiple covariance parameters).

To speed up search further, we restrict the second phase to split only abstractions that are likely to benefit from it. Phase two starts by computing $\mathcal{L}_{\mathbf{D}}(\mathcal{P})$ and recording the lowest-probability Gaussian variable of each abstraction. An abstraction $\mathbf{a}$ is only added to the splitting priority queue if this probability is lower than a threshold $\tau_{\mathbf{gauss}} = \exp(-1)$.

When both search phases finish, we run a final parameter optimization pass to optimize the σ's of all Gaussian distributions. This pass ensures that the final program generates only the appropriate variation (or no variation, if $\sigma \approx 0$) for each continuous variable.

Particle Filtering Parameter optimization is stochastic: each iteration draws random samples from the q_i's to compute $\tilde{\mathcal{L}}_{\mathbf{D}}(\mathcal{P})$. This stochasticity means that even with the careful design decisions described in the preceding paragraphs, optimization can still occasionally become stuck in spurious local optima. We use particle filtering to guard against this sporadic behavior [DDFG01]. During program learning, we run N copies of the structure search procedure in parallel. After each proposed split, we gather the current programs $\mathcal{P}$ of each search process and resample them according to their search objective scores $O_{\mathbf{D}}(\mathcal{P})$. If some search processes fall into spurious local optima, they will be outscored by others

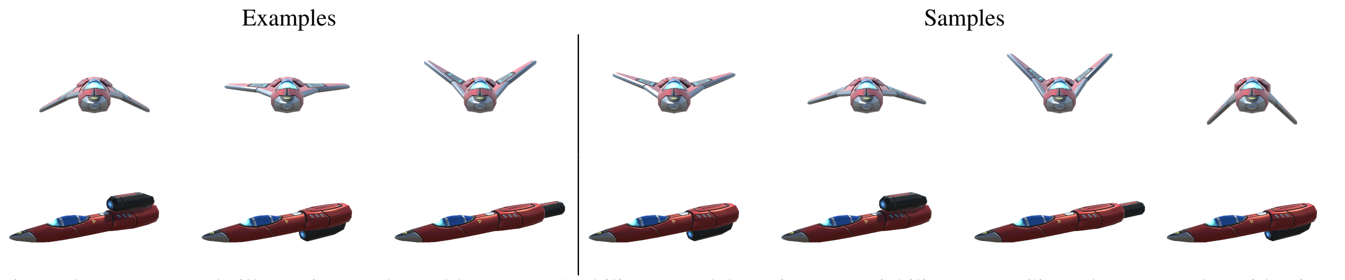

Figure 4: A toy example illustrating our learnable program's ability to model continuous variability. *(Top)*: Given three examples with wings in a range of configurations, the program learns to generate output samples which interpolate between and extrapolate from the example configurations. *(Bottom)*: The three examples put the engine in completely different configurations, and the program learns to use a separate abstraction for each of these configurations.

that do not and will tend to be thrown away during resampling. When all search processes terminate, we choose the one with the highest-scoring program as the final result.

7. Evaluation

In this section, we evaluate our method's ability to learn programs which plausibly generalize the input examples. As mentioned in the previous section, we have implemented a prototype program learning system in the WebPPL probabilistic programming language [GS14]. All experiments reported were run on Google Compute Engine virtual machine instances with 24 2.2GHz Intel Xeon cores and 48 GB RAM running Ubuntu 16.04.

Modeling continuous variability: toy examples Our learnable program representation is specifically designed to model variability in part transformations, so we first present toy examples which isolate and cleanly illustrate this behavior. Figure 4 Top shows three example models and representative samples generated by a program learned from them. The examples feature wings configured at 'keyframe' positions along a continuous range of angles from approximately -30° to 30°. As the samples show, the learned program can generate a smooth range of outputs which both interpolate between (first two samples) as well as extrapolate from (second two samples) those 'keyframes'. Figure 4 Bottom shows three examples which use the engine part in three distinct configurations, distinguished by large differences in transformations. Such large differences result in poor likelihoods when a single abstraction is used to model all three uses of the engine, leading the program to create a separate abstraction for each one. The resulting samples appropriately reproduce these distinct configurations, rather than (spuriously) attempting to blend between them.

Example dataset We built five sets of example models to use as test cases for our system: castles (*Castle*), ornamental graphic designs (*Ornament*), space stations (*SpaceStation*), and two types of spaceships (*StarSparrow* and *StriderOx*). Each set contains ten models. These models were built with modular 3D assets from the Unity Asset Store. Figure 5 shows the example models; this figure is best viewed on a high-resolution display.

Generating new objects The bottom row of each section in Figure 5 shows samples generated by a program learned from the examples in the row above, with certain images highlighted to illustrate particular behavior. The samples are stylistically similar to the examples and they exhibit structural and continuous variations not seen in the examples. For *StarSparrow* and *StriderOx*, the generated samples contain part combinations not seen in the examples. The highlighted *StarSparrow* sample illustrates continuous extrapolation, as the wings are bent at a lower angle than in any of the examples. In the highlighted *StriderOx* sample, the side-mounted, vertical engines are oriented differently than in any of the examples. The top-mounted fin is also placed in the center of the body, whereas it only occurs at the front or rear of the body in the examples—an example of continuous interpolation. The *SpaceStation* samples also show part variability, and the highlighted sample shows continuous extrapolation in the orientation of the top-most group of protrusions. In the hightlighted *Castle* samples, the 'outer wall' structure exhibits higher curvature than in the examples and also contains more wall segments. The learned *Ornament* program generates samples with a continuous range of bending angles. The highlighted sample illustrates recursive generalizationm, as the 'stem' structures are longer than in any of the examples. Additionally, the program correctly models parts which occur in distinct, different orientations (the stem pieces, the 'petals' at the end of each stem). Full-resolution versions of all of these images are available in the supplemental material.

Table 1 reports statistics for the example sets and results shown in Figure 5. For each of the example sets used, we report the number of unique parts used to construct the examples, the initial (i.e. before structure search) and final (i.e. after structure search) number of abstractions, the initial and final program learning objective score, and the computation time used by structure search. Due to our initialization strategy, the initial number of abstractions is influenced by the number of parts, with the 22-part *SpaceStation* example set having the highest number of initial abstractions. The *Ornament* example set has an especially low initial objective score. It frequently uses the same parts in multiple distinct orientations, leading to low probabilities under the transform distributions of the initial abstraction for those parts.

Structure search took on average ~ 1.7 hours to run for the example sets presented. There are several immediate opportunities for speeding up this process. First, we implemented our prototype in WebPPL, which is embedded in Javascript. An implementation based on a native-compiled numerics engine would be faster. Sec-

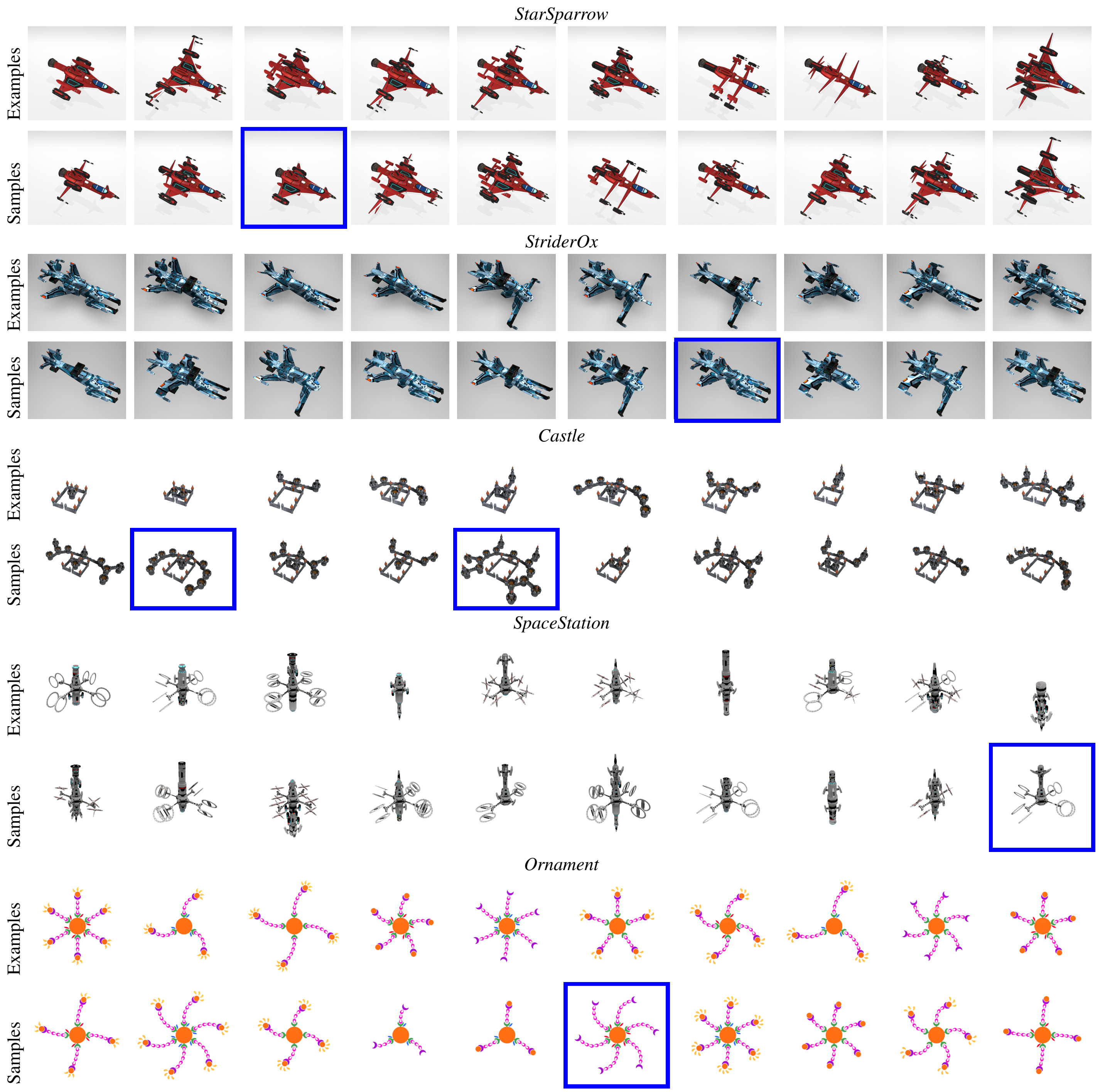

Figure 5: Five sets of example models and samples produced by programs learned from the examples. For each set, the top row shows the input examples and the bottom row shows generated samples. Samples which demonstrate interesting continuous and/or structural variability are highlighted in blue. Best viewed on a high-resolution display.

ond, our implementation re-optimizes all program parameters θ after each split proposal. However, it is only strictly necessary to re-optimize the parameters of the split abstractions and any abstractions which call them; an implementation which exploits this fact could further reduce its workload. Third, an implementation that uses the previous improvement could run fewer optimization iterations at each split proposal, since it would optimize a smaller, localized subset of program parameters. Finally, when evaluating

Example set	# Parts	$\lvert\mathcal{A}_0\rvert$	$O_{\mathbf{D}}(\mathcal{P}_0)$ $(\mathcal{L}_{\mathbf{D}}(\mathcal{P}_0) - C(\mathcal{P}_0))$	$\lvert\mathcal{A}^*\rvert$	$O_{\mathbf{D}}(\mathcal{P}^*)$ $(\mathcal{L}_{\mathbf{D}}(\mathcal{P}^*) - C(\mathcal{P}^*))$	Time (hrs)
StarSparrow	8	25	259.07 (332.07 − 73)	38	437.79 (538.79 − 101)	1.26
StriderOx	8	15	315.16 (360.16 − 45)	27	477.31 (552.31 − 75)	1.56
Castle	9	13	387.14 (426.14 − 39)	28	983.07 (1053.07 − 70)	1.82
SpaceStation	22	37	897.98 (1006.98 − 109)	53	967.64 (1103.64 − 136)	2.39
Ornament	7	14	-45.84 (−9.84 − 36)	26	453.75 (523.75 − 70)	1.65

Table 1: Summary statistics for the results shown in Figure 5. $\mathcal{P}_0 = (\mathcal{A}_0, \theta_0)$ is the initialization program, and $\mathcal{P}^* = (\mathcal{A}^*, \theta^*)$ is the final program returned by structure search. We report the number of parts used, the number of abstractions $\lvert\mathcal{A}\rvert$, the program learning objective (broken down into the data likelihood and complexity prior terms), and the wall-clock computation time used by structure search.

the data log likelihood and its gradient, our implementation processes all nodes of each example sequentially. An implementation which batches nodes by abstraction and evaluates batches in parallel would be faster still. We expect that such improvements should deliver an order-of-magnitude speedup, bringing the running time of structure search down to minutes.

Effect of each search phase Figure 6 shows representative examples of how each phase of structure search affects the behavior of the resulting program. Samples generated using the program $\mathcal{P}_0$ given by the initialization procedure tend to confuse different usages of the same part, e.g. the rear engine in the *StarSparrow* result which has inappropriate wings attached to it, or the *Ornament* result which generates an isolated 'flower' structure in place of the appropriate 'root' structure. The first phase of structure search, which considers only the discrete variables in the program, improves results significantly but can still fail to model distinct modes of continuous transformation present in the examples; this phase is analagous to discrete grammar-induction-based methods from prior work [TYK*]. In the *StarSparrow* examples, thrusters are mounted either above or below wings; discrete-only search results in a program that interpolates between these two configurations, sometimes producing samples with a thruster *inside* of a wing. Similarly, the *StriderOx* program fails to capture the three distinct configurations of the front-most wings, and the *Ornament* program has not identified the two different orientations of the pink 'stem' substructures. These issues are resolved by running the second, continuous-variable-aware phase of structure search. In some cases, the learned program exhibits acceptable behavior after the discrete phase (e.g. the *SpaceStation* result shown here). In such situations, the user could elect to terminate structure search early.

Effect of particle filter Figure 7 illustrates how the number of parallel search processes used (i.e. *particles*, in particle filtering terminology) affects the quality of the learned program for the *StriderOx* examples. Using only 1 particle results in high variance in the objective score $O_{\mathbf{D}}(\mathcal{P}^*)$ of the final learned program $\mathcal{P}^*$. The inset figure shows a sample from an incompletely-converged program learned in this way. Using a few more particles leads to much more reliable peformance: the results shown in this paper use 10, and the plot in Figure 7 suggests as few as 4 may be sufficient (and could be executed on a single multi-core CPU).

Variation and Generalization We next quantify our method's ability to capture the variation present in the input examples, i.e. ascertaining whether samples from learned programs exhibit as much variation as the examples. Specifically, we ask the question: on average, how similar are two objects drawn from the examples (s^{examps}) vs. two objects sampled from a learned program (s^{samps})? We measure this similarity in two different ways. First, we use a measure of the structural similarity between connectivity hierarchies ($s_{\mathbf{struc}}$) to capture large organizational variation in part structure. We define this measure as the size of the largest rooted subtree that two hierarchies have in common, normalized to $[0,1]$ by dividing by the maximum self-similarity of the two hierarchies. Second, we use a measure of visual similarity ($s_{\mathbf{vis}}$) to capture how variations affect an object's visual appearance. We define this measure using the Euclidean distance between two objects in a descriptor space computed by a state-of-the-art multi-view shape classifier [SMKL]. Distances are normalized to $[0,1]$ by dividing by the distance to the descriptor for the root part of the objects (since all objects have this sub-shape in common) and converted to similarities by taking one minus the distance.

Table 2 shows these average similarity measures for our five example sets. For $s_{\mathbf{struc}}$, none of the example sets showed a difference between s^{examps} and s^{samps} that was significant at a 95% confidence level. This result indicates that the examples and program samples have similiar structural variability. For $s_{\mathbf{vis}}$, only *Ornament* showed a significant difference, and the difference suggests that program samples exhibit slightly more variability than the examples.

We also quantify the extent to which learned programs generalize from the examples. As a first measure of generalization, Table 2 reports the percentage of generated object hierarchies that do not appear in the examples (% new structs). For all example sets, more than 90% of generated structures are new—very rarely does a learned program generate an exact structural duplicate of an example. Table 2 also reports the average similarity of a sample to its most-similar example ($s^{\max}$) for both the structural and visual similarity measures. These similarities are, as is to be expected, higher than the average similarity of sample pairs (s^{samps}). Nevertheless, they indicate that the learned program consistently generalizes beyond copy-and-paste of the input examples.

Perceptual study We also conducted a perceptual study in which objects generated by our programs were found indistinguishable from hand-created examples; see the supplemental material.

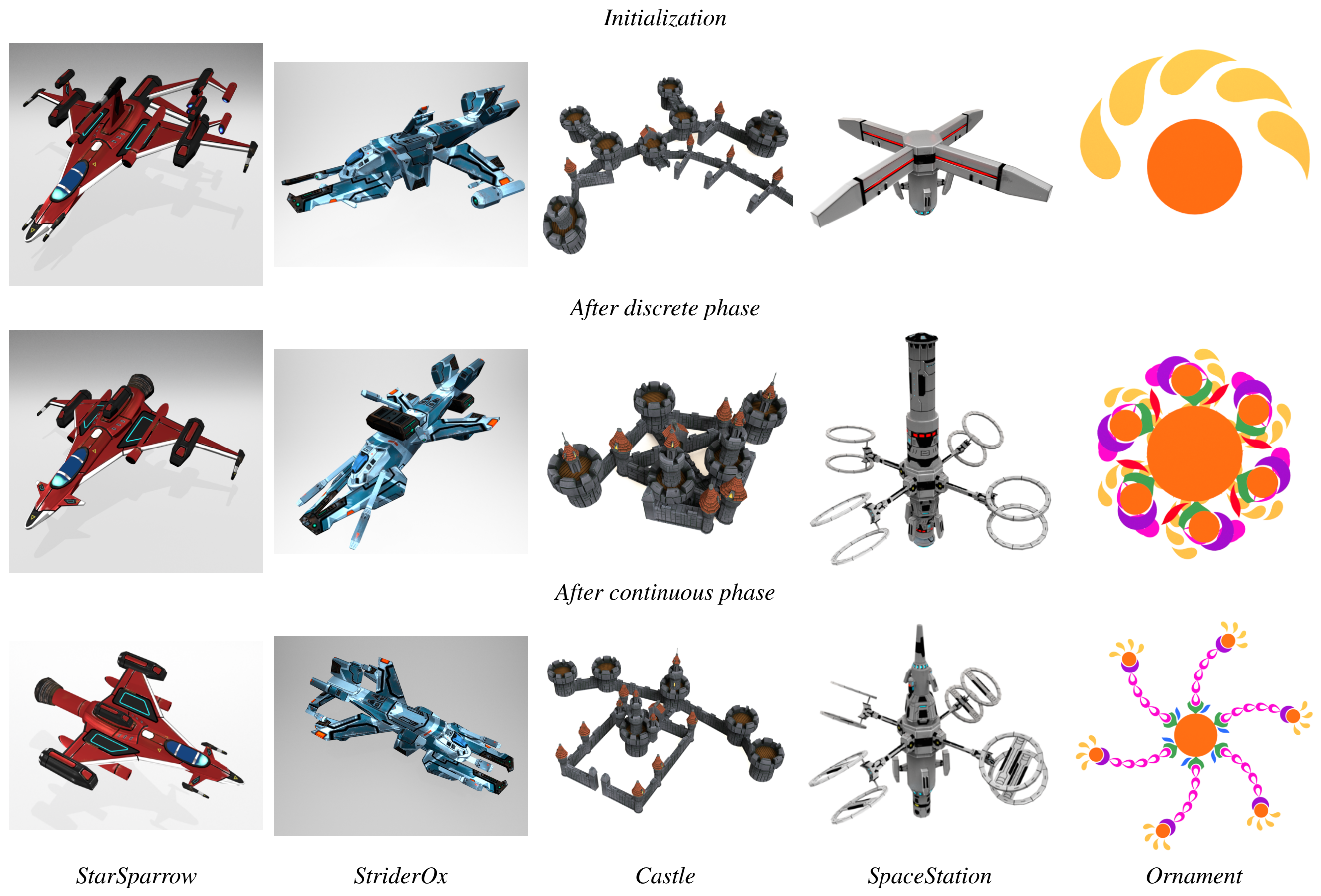

Figure 6: Representative samples drawn from the program with which we initialize structure search *(Top)*, the learned program after the first phase of structure search that considers only discrete variables *(Middle)*, and the learned program after the second phase of structure search that includes continuous variables *(Bottom)*. The program confuses different usages of the same part at initialization, and it fails to recognize distinct modes of spatial transformations after only the discrete phase of search.

Examples	$s^{\text{examps}}_{\textbf{struc}}$	$s^{\text{samps}}_{\textbf{struc}}$	$s^{\text{examps}}_{\textbf{vis}}$	$s^{\text{samps}}_{\textbf{vis}}$	% new structs	$s^{\max}_{\textbf{struc}}$	$s^{\max}_{\textbf{vis}}$
StarSparrow	0.35	0.34	0.44	0.39	98.5	0.71	0.55
StriderOx	0.49	0.48	0.40	0.36	98.8	0.77	0.50
Castle	0.52	0.50	0.46	0.40	97.8	0.76	0.58
SpaceStation	0.30	0.27	0.34	0.28	99.9	0.58	0.45
Ornament	0.28	0.24	0.60	0.55	90.3	0.73	0.68

Table 2: Measuring the variability and generalization capability of learned programs. We report the average similarity between two examples (s^{examps}) and two samples (s^{samps}) for measures of structural similarity ($s_{\textbf{struc}}$) and visual similarity ($s_{\textbf{vis}}$). *% new structs* gives the fraction of generated samples whose hierarchy structure does not occur in the examples, and $s^{\max}$ measures the average similarity of a generated sample to its most similar example.

8. Discussion and Future Work

This paper introduced a new method for learning procedural models of modular, part-based objects. Our learnable program representation models both the discrete, hierarchical structure of example objects as well as the variation in their continuous part arrangements. We developed an algorithm for learning this representation from examples which relies on a combination of combinatorial structure search and continuous parameter estimation via variational inference. Experiments demonstrated that programs learned using our approach can reliably generate a variety of new objects which are perceptually similar to the input examples.

The program representation we have developed uses simple neural networks to model the correlations between certain program variables. While these networks are flexible and efficiently learnable, they are unfortunately not very *interpretable*: a human reader inspecting a network's learned weights cannot easily determine

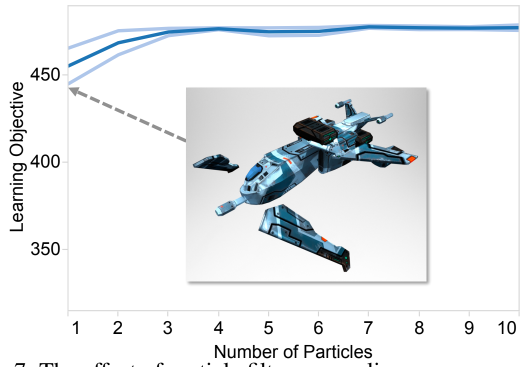

Figure 7: The effect of particle filter resampling on program learning for *StriderOx*. X-axis is the number of particles used; y-axis is the objective score $O_{\mathbf{D}}(\mathcal{P}^*)$ of the final learned program $\mathcal{P}^*$. The lowest y-value shown is $O_{\mathbf{D}}(\mathcal{P}_0)$, i.e. the score of the initialization program. Light blue lines visualize the 95% confidence interval. The inset figure shows a representative sample from a program learned using only 1 particle.

what function the network implements or how to modify that function. In an ideal world, procedural models learned from examples would be fully human-readable and human-editable. Toward this end, it may be possible to replace our programs' neural network components with short, readable code snippets generated by a constraint-based program synthesizer [SL08]. NADEs may be especially amenable to this type of translation, as (like programs) they are directed, sequential generative models.

Our program representation makes similar independence assumptions as a probabilistic context-free grammar, namely the Markov assumption that the probabilities of generating a node's children depend only on that node and not any other ancestors or non-descendants. While the slot-based representation allows for modeling dependencies between children, this Markov assumption means that our programs cannot capture long-range dependencies that cut across hierarchy levels. Future work could investigate how to extend the program to encode such dependencies in a manner that still supports learning from a small number of examples.

Our approach assumes that input examples have a tree-structured connectivity graph. Not all objects naturally admit such a decomposition, e.g. certain types of chairs have cyclical part connectivity. Such objects can be represented by an alternative hierarchical form in which connected or symmetric parts are successively merged together [WXL*]. Recently-developed methods based on deep recursive autoencoders have shown promising ability to generate such hierarchies, though they require significant data to do so [LXC*]. Further work is needed to develop procedural representations that can learn from a small number of such hierarchies, or that can learn to generate cyclic graphs directly.

While we have focused on the setting of learning from a small number of examples produced by a single user, the methods we have presented are also extensible to larger-data settings. The parameter optimization scheme that forms the backbone of our learning procedure is based on stochastic gradient descent and is thus scalable to large datasets by minibatching. As it becomes intractable to maintain individual q_i distributions for each node in a large dataset, these distributions would need to be replaced with *inference networks* that estimate abstraction probabilities for each node based on features of the node and its surrounding context [MG]. Recursive neural networks could work well here.

In this work, we focused on modeling objects with parts arranged by rigid transformation. However, the neural-network components we use to model these transformations are flexible enough to support other types of continuous variations on object parts. We are interested in applications of our method to model objects whose components can undergo articulation, non-rigid deformation, or other forms of parameterized geometric variability.

More generally, the machinery developed in this paper—combinatorial search over a space of programs which employ simple function approximators—should be applicable to modeling other types of graphical content beyond part-based objects. Architectural forms, such as building masses and facades, are one such possible application domain. We are excited to see what other kinds of procedural models can be learned using techniques like ours.

Acknowledgments

Thanks to Noah Goodman, Pat Hanrahan, and Matt Fisher for helpful discussions, and to Matt Fisher, Manolis Savva, and Kai Wang for feedback on earlier drafts of this paper. The *StarSparrow*, *StriderOx*, and *SpaceStation* assets were created by Unity Asset Store publisher Ebal Studios, and the *Castles* assets were created by publisher LowlyPoly. The *Flowers* assets are derived from images by Shutterstock publisher Bariskina. This material is based on research sponsored by DARPA under agreement number FA8750-14-2-0009. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[BWS] Bokeloh M., Wand M., Seidel H.: A connection between partial symmetry and inverse procedural modeling. In *SIGGRAPH 2010*. 2

[CF16] Chua J., Felzenszwalb P. F.: Scene Grammars, Factor Graphs, and Belief Propagation. *CoRR arXiv:1606.01307* (2016). 2

[CKGK] Chaudhuri S., Kalogerakis E., Guibas L., Koltun V.: Probabilistic Reasoning for Assembly-Based 3D Modeling. In *SIGGRAPH 2011*. 2

[DDFG01] Doucet A., De Freitas N., Gordon N. (Eds.): *Sequential Monte Carlo Methods in Practice*. Springer, 2001. 7

[DLC*] Dang M., Lienhard S., Ceylan D., Neubert B., Wonka P., Pauly M.: Interactive Design of Probability Density Functions for Shape Grammars. In *SIGGGRAPH Asia 2015*. 2

[ESLT] Ellis K., Solar-Lezama A., Tenenbaum J. B.: Unsupervised Learning by Program Synthesis . In *NIPS 2015* . 2

[FW16] Fan L., Wonka P.: A probabilistic model for exteriors of residential buildings. *ACM Transactions on Graphics 35*, 5 (2016), 155:1–155:13. 2

[GLSM] Gu S., Levine S., Sutskever I., Mnih A.: MuProp: Unbiased Backpropagation for Stochastic Neural Networks. In *ICLR 2016*. 3

[Gru07] Grunwald P. D.: *The Minimum Description Length Principle*. The MIT Press, 2007. 6

[GS14] Goodman N. D., Stuhlmüller A.: The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014. Accessed: 2016-11-2. 6, 8

[HKYM16] Huang H., Kalogerakis E., Yumer M. E., Mech R.: Shape Synthesis from Sketches via Procedural Models and Convolutional Networks. *IEEE Transactions on Visualization and Computer Graphics* (2016). 2

[HM98] Hogenhout W. R., Matsumoto Y.: A fast method for statistical grammar induction. *Natural Language Engineering 4*, 3 (1998), 191âĂŞ209. 7

[HSG11] Hwang I., Stuhlmüller A., Goodman N. D.: Inducing Probabilistic Programs by Bayesian Program Merging. *CoRR arXiv:1110.5667* (2011). 2, 3

[JTRS] Jain A., Thormhlen T., Ritschel T., Seidel H.-P.: Exploring Shape Variations by 3D-Model Decomposition and Part-based Recombination . In *Eurographics 2012* . 2

[KB] Kingma D. P., Ba J.: Adam: A Method for Stochastic Optimization. In *ICLR 2015*. 6

[KCKK] Kalogerakis E., Chaudhuri S., Koller D., Koltun V.: A Probabilistic Model for Component-based Shape Synthesis. In *SIGGRAPH 2012*. 2

[KW] Kingma D. P., Welling M.: Auto-Encoding Variational Bayes. In *ICLR 2014*. 3

[LCK*] Liu T., Chaudhuri S., Kim V. G., Huang Q., Mitra N. J., Funkhouser T.: Creating Consistent Scene Graphs Using a Probabilistic Grammar. In *SIGGRAPH Asia 2014*. 2, 3

[LST15] Lake B., Salakhutdinov R., Tenenbaum J. B.: Human-level concept learning through probabilistic program induction . *Science 350*, 6266 (2015), 1132–1138. 2

[LVW*] Liu H., Vimont U., Wand M., Cani M.-P., Hahmann S., Rohmer D., Mitra N. J.: Replaceable Substructures for Efficient Part-Based Modeling . In *Eurographics 2015* . 2

[LXC*] Li J., Xu K., Chaudhuri S., Yumer E., Zhang H., Guibas L.: GRASS: Generative Recursive Autoencoders for Shape Structures. In *SIGGRAPH 2017*. 2, 12

[MG] Mnih A., Gregor K.: Neural Variational Inference and Learning in Belief Networks. In *ICML 2014*. 3, 12

[MVG] Martinovic A., Van Gool L.: Bayesian Grammar Learning for Inverse Procedural Modeling. In *CVPR 2013*. 2, 3, 7

[MWH*] Müller P., Wonka P., Haegler S., Ulmer A., Van Gool L.: Procedural Modeling of Buildings. In *SIGGRAPH 2006*. 1

[NGDA*] Nishida G., Garcia-Dorado I., Aliaga D. G., Benes B., Bousseau A.: Interactive Sketching of Urban Procedural Models. In *SIGGRAPH 2016*. 2

[PJM] Prusinkiewicz P., James M., Měch R.: Synthetic Topiary. In *SIGGRAPH 1994*. 1

[RGB] Ranganath R., Gerrish S., Blei D. M.: Black Box Variational Inference. In *AISTATS 2014*. 2, 3, 6

[RHG16] Ritchie D., Horsfall P., Goodman N. D.: Deep Amortized Inference for Probabilistic Programs. *CoRR arXiv:1610.05735* (2016). 3, 6

[RMGH] Ritchie D., Mildenhall B., Goodman N. D., Hanrahan P.: Controlling Procedural Modeling Programs with Stochastically-Ordered Sequential Monte Carlo. In *SIGGRAPH 2015*. 2

[RTHG] Ritchie D., Thomas A., Hanrahan P., Goodman N. D.: Neurally-Guided Procedural Models: Amortized Inference for Procedural Graphics Programs using Neural Networks. In *NIPS 2016*. 2

[SBM*] Stava ., Benes B., Mech R., Aliaga D. G., Kristof P.: Inverse Procedural Modeling by Automatic Generation of L-systems. In *Eurographics 2010*. 2

[SHWA] Schulman J., Hess N., Weber T., Abbeel P.: Gradient Estimation Using Stochastic Computation Graphs. In *NIPS 2015*. 3, 6

[SL08] Solar Lezama A.: *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, 2008. 12

[SMKL] Su H., Maji S., Kalogerakis E., Learned-Miller E. G.: Multi-view convolutional neural networks for 3d shape recognition. In *ICCV 2015*. 10

[SO94] Stolcke A., Omohundro S.: Inducing probabilistic grammars by bayesian model merging. In *Proc. International Colloqium on Grammatical Inference and Applications* (1994), pp. 106–118. 7

[SPK*14] Stava O., Pirk S., Kratt J., Chen B., Měch R., Deussen O., Benes B.: Inverse Procedural Modelling of Trees. *Computer Graphics Forum 33*, 6 (2014). 2

[TLL*11] Talton J. O., Lou Y., Lesser S., Duke J., Měch R., Koltun V.: Metropolis Procedural Modeling. *ACM Transactions on Graphics 30*, 2 (2011). 2

[TYK*] Talton J., Yang L., Kumar R., Lim M., Goodman N., Měch R.: Learning Design Patterns with Bayesian Grammar Induction. In *UIST 2012*. 1, 2, 3, 7, 10

[UCG*16] Uria B., Côté M.-A., Gregor K., Murray I., Larochelle H.: Neural autoregressive distribution estimation. *Journal of Machine Learning Research 17*, 205 (2016), 1–37. 5

[Uni] Unity Technologies: Unity Asset Store. https://assetstore.unity.com. Accessed: 2017-09-18. 1

[VGDA*] Vanegas C. A., Garcia-Dorado I., Aliaga D. G., Benes B., Waddell P.: Inverse Design of Urban Procedural Models. In *SIGGRAPH Asia 2012*. 2

[WXL*] Wang Y., Xu K., Li J., Zhang H., Shamir A., Liu L., Cheng Z., Xiong Y.: Symmetry Hierarchy of Man-Made Objects. In *Eurographics 2011*. 3, 12

[WYD*] Wu F., Yan D.-M., Dong W., Zhang X., Wonka P.: Inverse Procedural Modeling of Facade Layouts. In *SIGGRAPH 2014*. 2

[WZS] Wong M. T., Zongker D. E., Salesin D. H.: Computer-generated Floral Ornament. In *SIGGRAPH 1998*. 1