ShapeAssembly: Learning to Generate Programs for 3D Shape Structure Synthesis

R. KENNY JONES, Brown University
THERESA BARTON, Brown University
XIANGHAO XU, Brown University
KAI WANG, Brown University
ELLEN JIANG, Brown University
PAUL GUERRERO, Adobe Research
NILOY J. MITRA, University College London, Adobe Research
DANIEL RITCHIE, Brown University

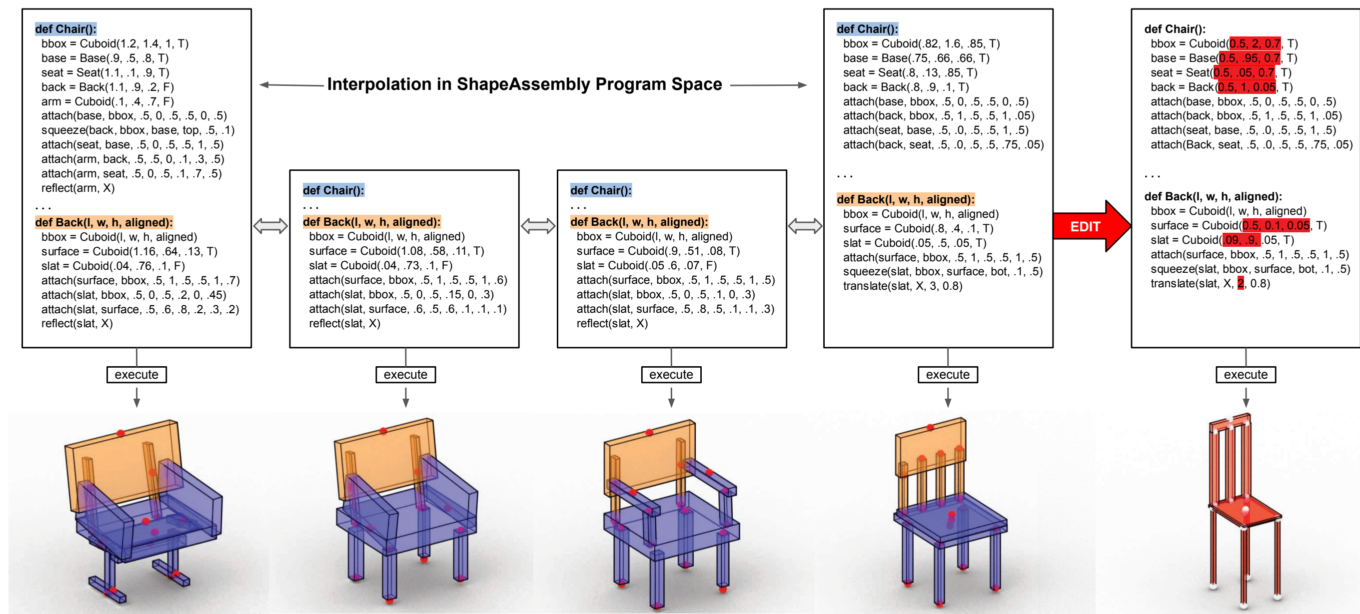

Fig. 1. We present a deep generative model which learns to write novel programs in ShapeAssembly, a domain-specific language for modeling 3D shape structures. Executing a ShapeAssembly program produces a shape composed of a hierarchical connected assembly of part proxies cuboids. Our method develops a well-formed latent space that supports interpolations between programs. Above, we show one such interpolation, and also visualize the geometry these programs produce when executed. In the last column, we manually edit the continuous parameters of a generated program, in order to produce a variant geometric structure with new topology.

Manually authoring 3D shapes is difficult and time consuming; generative models of 3D shapes offer compelling alternatives. Procedural representations are one such possibility: they offer high-quality and editable results but are difficult to author and often produce outputs with limited diversity. On the other extreme are deep generative models: given enough data, they can learn to generate any class of shape but their outputs have artifacts and the representation is not editable.

In this paper, we take a step towards achieving the best of both worlds for novel 3D shape synthesis. First, we propose ShapeAssembly, a domain-specific "assembly-language" for 3D shape structures. ShapeAssembly programs construct shape structures by declaring cuboid part proxies and attaching them to one another, in a hierarchical and symmetrical fashion. ShapeAssembly functions are parameterized with continuous free variables, so that one program structure is able to capture a family of related shapes.

Authors' addresses: R. Kenny Jones, Brown University; Theresa Barton, Brown University; Xianghao Xu, Brown University; Kai Wang, Brown University; Ellen Jiang, Brown University; Paul Guerrero, Adobe Research; Niloy J. Mitra, University College London, Adobe Research; Daniel Ritchie, Brown University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.

0730-0301/2020/12-ART234 $15.00
https://doi.org/10.1145/3414685.3417812

We show how to extract ShapeAssembly programs from existing shape structures in the PartNet dataset. Then, we train a deep generative model, a hierarchical sequence VAE, that learns to write novel ShapeAssembly programs. Our approach leverages the strengths of each representation: the program captures the subset of shape variability that is interpretable and editable, and the deep generative model captures variability and correlations across shape collections that is hard to express procedurally.

We evaluate our approach by comparing the shapes output by our generated programs to those from other recent shape structure synthesis models. We find that our generated shapes are more plausible and physically-valid than those of other methods. Additionally, we assess the latent spaces of these models, and find that ours is better structured and produces smoother interpolations. As an application, we use our generative model and differentiable program interpreter to infer and fit shape programs to unstructured geometry, such as point clouds.

CCS Concepts: • **Computing methodologies → Neural networks**; **Latent variable models**; **Shape analysis**.

Additional Key Words and Phrases: Shape analysis, shape synthesis, generative models, deep learning, procedural modeling, neurosymbolic models

ACM Reference Format:
R. Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. 2020. ShapeAssembly: Learning to Generate Programs for 3D Shape Structure Synthesis. *ACM Trans. Graph.* 39, 6, Article 234 (December 2020), 20 pages. https://doi.org/10.1145/3414685.3417812

1 INTRODUCTION

3D models of human-made objects are more in-demand than ever. In addition to the traditional drivers of demand in computer graphics (visual effects, animation, games), new applications in artificial intelligence increasingly benefit from or even require high-quality 3D objects, such as producing synthetic training imagery for computer vision systems [Kar et al. 2019; Richter et al. 2016; Zhang et al. 2017] or training robots to perform tasks in virtual environments [Abbatematteo et al. 2019; Kolve et al. 2017; Savva et al. 2019; Xia et al. 2018]. Despite the growing demand, the craft of 3D modeling largely remains as difficult and time-consuming as it has ever been. The time and expertise required to create 3D content by hand will not scale to these demands.

One promising way out of this conundrum is the development of *generative models* of 3D shapes, i.e. procedures which can be executed to generate novel shapes within some class [Müller et al. 2006; Parish and Müller 2001; Prusinkiewicz and Lindenmayer 1996]. An ideal generative model would produce plausible output geometry, capture a wide range of shape variations, and use an interpretable representation which a user could subsequently manipulate and edit. Unfortunately, no existing shape generative model achieves all of these properties. On the one hand are *procedural models*: structured computer programs which produce geometry when executed. Procedural models can produce high-quality geometry, and their program-based representation makes them interpretable and editable to users with some programming background. However, authoring a good procedural model from scratch is difficult (arguably at least as difficult as modeling an object by hand), and the amount of shape variation captured by a single procedural model is limited (e.g., it is difficult to write one program that can model all types of cars). On the other hand are data-driven generative models, particularly *deep generative models*: neural networks which learn how to generate 3D shapes from data [Fan et al. 2017; Groueix et al. 2018; Li et al. 2017; Mo et al. 2019a; Wu et al. 2016]. Deep generative models capture variability with little human effort: given enough training data, they can in theory learn to generate any class of shape. Since they lack the strict semantics of programs, however, their outputs often exhibit "noise" artifacts such as incomplete geometry and floating parts. Additionally, the representations they learn are typically inscrutable to people, making them hard to edit or manipulate in predictable ways.

Our insight, in this work, is that these two approaches have complementary strengths: deep generative models are efficient to create and excel at broad-scale variability, and procedural models produce high-quality geometry by construction and better facilitate editing for fine-scale variability. We take a first step toward achieving the best of both worlds by integrating these two approaches into a single pipeline: a deep generative model that *learns to write programs*, which, when executed, themselves output 3D geometry. We hypothesize that going through this intermediate program representation produces a generative model with a smoother latent space, whose outputs are more likely to be physically valid, compact, and editable.

As the motivating applications mentioned earlier demand 3D models of human-made objects, we focus on generating novel part-based shape structures in this paper. We introduce ShapeAssembly, an "assembly language" for 3D shape structures. In ShapeAssembly, shape structures are represented by hierarchical assemblies of connected parts, where leaf-level parts are approximated by a bounding cuboid (a similar representation as the ones used by PartNet [Mo et al. 2019b] and StructureNet [Mo et al. 2019a]); these hierarchical cuboid structures can then be used to condition the generation of shape surface geometry in the form of e.g. point clouds. A ShapeAssembly program constructs a shape by declaring cuboids, iteratively attaching them to one another, and specifying symmetric repetitions of connected cuboid assemblies. The dimensions of these cuboids and the positions of these attachments are a program's parameters; manipulating them allows for exploring a family of related shapes. Furthermore, our interpreter for executing ShapeAssembly programs is fully differentiable, meaning it is possible to compute gradients of a program's output geometry with respect to its continuous parameters. Figure 1 shows some example hierarchical ShapeAssembly programs and the output shapes they produce.

While ShapeAssembly programs produce valid geometry under a range of parameter values, they do not exhibit *structural* variability, and authoring them from scratch still takes time. Thus, we train a neural network to write a variety of ShapeAssembly programs for us. Using programs we extract from a shape dataset, we train a hierarchical sequence VAE which outputs hierarchical ShapeAssembly programs. Each node in the hierarchy uses a recurrent language model to generate the program text at that level, and to decide which cuboids should be expanded into subroutine calls. Furthermore, the well-defined semantics of ShapeAssembly allow us to identify semantically-invalid programs and modify the generator such that it never produces them. The programs shown in Figure 1 were written by our generative model, by decoding code vectors

along a straight line in its latent space. We show that this generative model indeed learns to generate plausible, novel shape programs that were never seen its the training set.

We evaluate our approach by comparing it to other recently-proposed generative models of 3D shape structure along several axes including plausibility, diversity, complexity, and physical validity. We find that our generated shapes are both more plausible and more physically-valid than those of other methods. Additionally, we assess the latent spaces of these models, and find that ours is better structured and produces smoother interpolations, both in terms of geometric and structural continuity. As a bonus, we also show that ShapeAssembly's decoder does a better job of fitting programs to unstructured point clouds while also maintaining physical validity, and that this performance difference is magnified by optimizing the program fit via our differentiable interpreter.

In summary, our contributions are:

(i) The ShapeAssembly language and its differentiable interpreter, allowing the procedural specification of shape structures represented as connected part assemblies.
(ii) A deep generative model for ShapeAssembly programs, coupling the ease-of-training and variability of neural networks with the precision and editability of procedural representations.

Code and data used for all of our experiments can be found at https://github.com/rkjones4/ShapeAssembly .

2 RELATED WORK

Deep Generative Models of 3D Shapes. Recent years have seen an explosion of activity in applying deep generative models to 3D shape generation. Some of the earliest approaches generated shapes as 3D occupancy grids [Wu et al. 2016; Z. Wu 2015]; later work has explored generative representations of point clouds [Fan et al. 2017], 2D surface patches [Groueix et al. 2018], and implicit surfaces [Chen et al. 2019; Chen and Zhang 2019; Michalkiewicz et al. 2019; Park et al. 2019]. Our approach is more closely related to generative models of *part-based* shapes, wherein a complete object is synthesized by generating and assembling multiple subparts. These include approaches for iteratively adding parts to partially-complete shapes [Sung et al. 2017], generating symmetry hierarchies [Li et al. 2017], composing parts from two different shapes [Zhu et al. 2018], and generating hierarchical connectivity graphs [Mo et al. 2019a]. Our method is different in that we do not aim to learn a generative model that outputs a single shape; rather, ours outputs a procedural program which then itself generates a related family of shapes.

Procedural and Inverse Procedural Modeling. There is a rich history of methods for procedural modeling in computer graphics: especially noteworthy examples include its use in modeling plants [Prusinkiewicz and Lindenmayer 1996] and urban environments [Müller et al. 2006; Parish and Müller 2001]. Most procedural modeling systems use some form of (context-free) grammar, i.e., a recursive string re-writing system (which may be interpreted as e.g., recursively splitting a spatial domain, in the case of shape grammars). Additionally, attachment-based grammars of part assemblies have been used to aid in the structural analysis of shapes [Lau et al. 2011]. Our procedural representation is fundamentally different: we use an imperative language which iteratively constructs shapes via declaring and then connecting parts represented as simple proxy geometry. Also related to our work is the line of research on *inverse* procedural modeling, i.e., inferring a procedural model from a set of examples [Hwang et al. 2011; Martinovic and Van Gool 2013; Nishida et al. 2018, 2016; Ritchie et al. 2018; Talton et al. 2012]. These methods all strive to infer an interpretable, stochastic program which generates multiple output shapes. In contrast, we represent shapes via *deterministic* programs, and then we use a stochastic neural network to generate those programs.

Visual Program Induction. Another related line of work to ours is *visual program induction* (VPI): the practice of inferring a program which describes a single visual entity, such as a 3D shape. We address a fundamentally different problem: training a generative model to generate *novel* 3D shape programs from scratch. We do use a VPI-like process as a subroutine, to convert every shape in a large dataset into training programs for our generative model. Prior work in this area can be roughly divided into two categories: methods that assume that clean, segmented geometry is available and then use geometric heuristics to infer a program [Demir et al. 2016; Stava et al. 2010], and methods which use learning or optimization to operate directly on "raw" visual inputs such as images and occupancy grids [Du et al. 2018; Ellis et al. 2019, 2018; Liu et al. 2019; Sharma et al. 2018; Tian et al. 2019; Zhou et al. 2019; Zou et al. 2017]. Our approach to extracting programs from shapes to formulate training data is more similar to the former.

One could consider solving our problem of novel shape program generation by first generating novel 3D *shapes* with an existing shape generative model and then using a VPI-like system to infer a program describing that shape. However, as we will later show, the programs produced by such a process are less clean and editable than ones generated by our model; furthermore, training to generate programs rather than shapes directly actually produces a better-structured latent space.

Our complete pipeline of using a neural network to generate a program and then using that program to generate the ultimate output is also related to work in visual question answering which uses neural networks to generate a "query program" for each question which then analyzes the input image and produces an answer [Johnson et al. 2017]. It is also related to work that tries to combine the advantages of neural guidance with symbolic search for performing inference over structured domains [Lu et al. 2019].

Our work is the first to train a deep generative model to produce *novel* shape programs from scratch, each of which outputs a parametric family of related 3D shapes. This pipeline combines the advantages of neural and procedural shape modeling.

3 APPROACH

Figure 2 shows our overall pipeline. Our approach is divided into the following stages:

Input. Our pipeline takes as input a large dataset of *hierarchical 3D part graphs* [Mo et al. 2019a,b]. This is a shape representation in

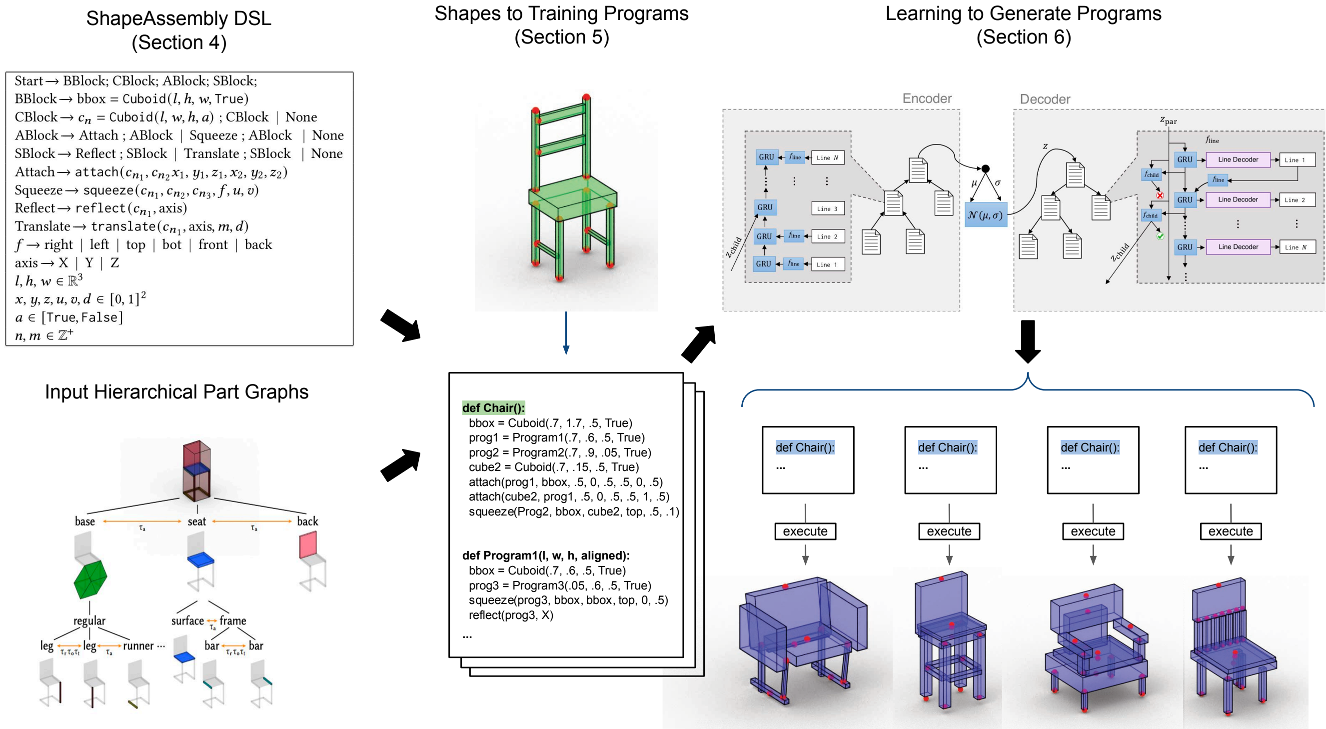

Fig. 2. Our pipeline for generating 3D shape structure programs. We first define a DSL language for 3D shapes, ShapeAssembly . Then, given a dataset of hierarchical part graphs, we extract ShapeAssembly programs from them. Finally, we use these programs as training data for a deep generative model. Our method learns to generate novel program instances that can be executed to produce complex and interesting 3D shape structures.

which each node represents a part in a shape consisting of an assembly of parts. Nodes are connected via edges that denote physical part attachments. They can also be connected via parent-child edges that denote hierarchy relationships (i.e., that one part is composed of several other smaller parts). At the leaf level of this hierarchy, atomic parts are represented by cuboid proxy geometry (typically computed from minimum-volume bounding boxes of more detailed part meshes).

Defining a DSL for connected, hierarchical shapes. To represent shapes as programs, we introduce a domain-specific language (DSL). Since our input shapes are characterized by graphs of parts, where graph edges denote physical part connections, we introduce a DSL based around declaring parts and then attaching them to one another. We call this language ShapeAssembly (as in, an "assembly language" for shapes). Section 4 describes the language.

Creating a dataset of shape-program pairs. Given the language described above, we present a method for finding programs that represent the shapes in our dataset. In our procedure, we first extract the program content based on a combination of data cleaning and geometric analysis. Then, we create canonical programs through a series of ordering and filter steps. Section 5 describes this procedure in more detail.

Learning to generate programs. Finally, we treat the programs extracted from each shape as training data for a generative model. Section 6 describes our deep generative model's architecture, the procedure we use to train it, and how we sample from it to synthesize new programs, which when executed produce novel shape structures.

4 AN ASSEMBLY LANGUAGE FOR SHAPES

Our goal in this section is to define a domain-specific language for shapes which are specified as connected assemblies of parts. As we focus on the problem of shape structure synthesis, cuboids, serving as part proxy geometry, are the only data type in our language. In Section 7, we show how to use other existing techniques to convert these proxies into surface geometry.

The primary operation in the language is attaching these cuboids together. Attachment turns out to be a very powerful and flexible operation. In fact, our language does not include any operations for explicitly positioning or orienting cuboids: all of this is accomplished via attachment operations. Additionally, the language includes higher-level *macros* that capture more complex spatial relationships, such as symmetry. At execution time, each macro is expanded into a series of cuboid declarations and attachment operations.

We call this DSL ShapeAssembly, because it is an "assembly language for shapes": a low-level language for creating shapes, in which shapes are created by assembling parts. Table 1 shows the grammar for ShapeAssembly, and Figure 3 shows an annotated hierarchical program along with its executed 3D shape.

A ShapeAssembly program consists of four main blocks:

- **BBlock:** Declares a non-visible bounding volume of the overall shape. This bounding volume is treated as a physical entity to which other parts can be connected.
- **CBlock:** Declares all the cuboid part proxies that will be used by the remainder of the program. The `Cuboid` command takes in l, w, h parameters that control the starting dimensions of the part, and an aligned flag a that specifies if the part has the same orientation as its bounding volume.
- **ABlock:** Connects cuboids by iteratively attaching them to one another. The `attach` command takes in two cuboids, c_{n1}, c_{n2}, and attaches the point (x_1, y_1, z_1) in the local coordinate frame of c_{n1} with the point (x_2, y_2, z_2) in the local coordinate frame of c_{n2}. The `squeeze` macro expands into two `attach` statements, such that c_{n1} is placed in-between c_{n2} and c_{n3} along the specified face f at the face-coordinate position (u, v).
- **SBlock:** Generates symmetry groups by instantiating additional `Cuboid` and attach commands. The `reflect` macro reflects cuboid c_n over axis *axis* of the bounding volume. The `translate` macro creates a translational symmetry group starting at c_n with m additional members along axis a of the bounding volume that ends distance d away.

Semantics. ShapeAssembly has imperative semantics: every line of the program immediately takes effect and alters the state of the shape being constructed. Figure 4 shows an example of a simple shape being imperatively constructed. Declaring a cuboid instantiates a new piece of cuboid geometry with the requested dimensions, centered at the origin. Invoking the `attach` command alters the cuboid, potentially translating, rotating, or resizing it in order to

```
1.  def Chair():
2.    bbox = Cuboid(1, 1.5, .8, True)
3     base = Base(.8, .5, .8, True)
4.    cube1 = Cuboid(.8, .1, .8, True)
5.    back = Back(.9, .8, .07, True)
6.    attach(base, bbox, .5, 0, .5, .5, 0, .5)
7.    attach(cube1, base, .5, 0, .5, .5, 1, .5)
8.    squeeze(back, bbox, cube1, top, .5, .05)

9.  def Base(l, w, h, aligned):
10.   bbox = Cuboid(l, w, h, aligned)
11.   cube0 = Cuboid(.2, .5, .2, True)
12    cube1 = Cuboid(.2, .5, .2, True)
13.   squeeze(cube0, bbox, bbox, top, .1, .1)
14.   squeeze(cube1, bbox, bbox, top, .1, .8)
15.   reflect(cube0, X)
      reflect(cube1, X)

17. def Back(l, w, h, aligned):
18.   bbox = Cuboid(l, w, h, aligned)
19.   cube0 = Cuboid(.9, .4, .07, True)
20.   cube1 = Cuboid(.1, .4, .05, True)
21.   attach(cube0, bbox, .5, 1, .5, .5, 1, .5)
22.   squeeze(cube1, bbox, cube0, bot, .3, .5)
23.   translate(cube1, X, 2, .5)
```

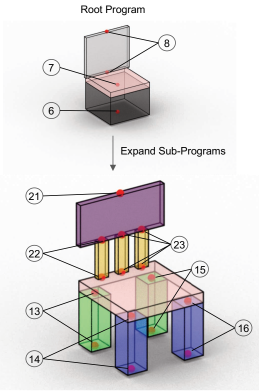

Fig. 3. An example ShapeAssembly program and the shape that it generates. Parts are colored according to the line of the program which instantiates them, and attachment points are numbered accordingly. In the top shape, we show the executed Chair program without hierarchy. In the bottom shape, we show the Chair program executed hierarchically with its sub-programs (Base and Back). For instance, the light grey back part is expanded into the purple back surface and gold slats.

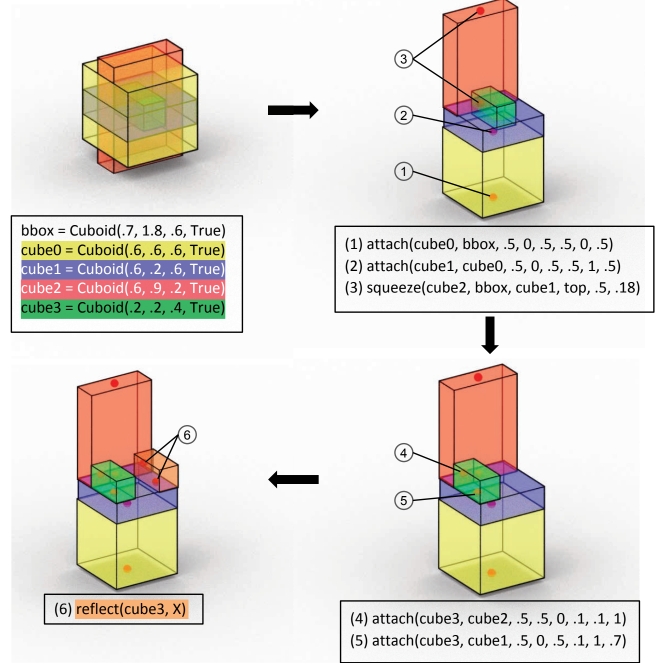

Fig. 4. An illustration of how the ShapeAssembly interpreter incrementally constructs shapes by imperatively executing program commands. Cuboids are instantiated at the origin and are moved through attachment. Notice how the `reflect` command in line 6 acts as a macro function, creating a new cuboid and two new attachments.

Table 1. The grammar for ShapeAssembly, our low-level domain-specific "assembly language" for shape structure. A program consists of `Cuboid` statements which instantiate new geometry and `attach` statements which connect these geometries together at specified points on their surfaces. Macro functions (`reflect`, `translate`, `squeeze`) form complex spatial relationships by expanding into multiple `Cuboid` and `attach` statements.

Start → BBlock; CBlock; ABlock; SBlock;
BBlock → bbox = `Cuboid`$(l, h, w,$ `True`$)$
CBlock → c_n = `Cuboid`(l, w, h, a) ; CBlock | None
ABlock → Attach ; ABlock | Squeeze ; ABlock | None
SBlock → Reflect ; SBlock | Translate ; SBlock | None
Attach → `attach`$(c_{n_1}, c_{n_2}, x_1, y_1, z_1, x_2, y_2, z_2)$
Squeeze → `squeeze`$(c_{n_1}, c_{n_2}, c_{n_3}, f, u, v)$
Reflect → `reflect`$(c_n,$ axis$)$
Translate → `translate`$(c_n,$ axis$, m, d)$
f → right | left | top | bot | front | back
axis → X | Y | Z
$l, h, w \in \mathbb{R}^+$
$x, y, z, u, v, d \in [0, 1]^2$
$a \in$ [`True`, `False`]
$n, m \in \mathbb{Z}^+$

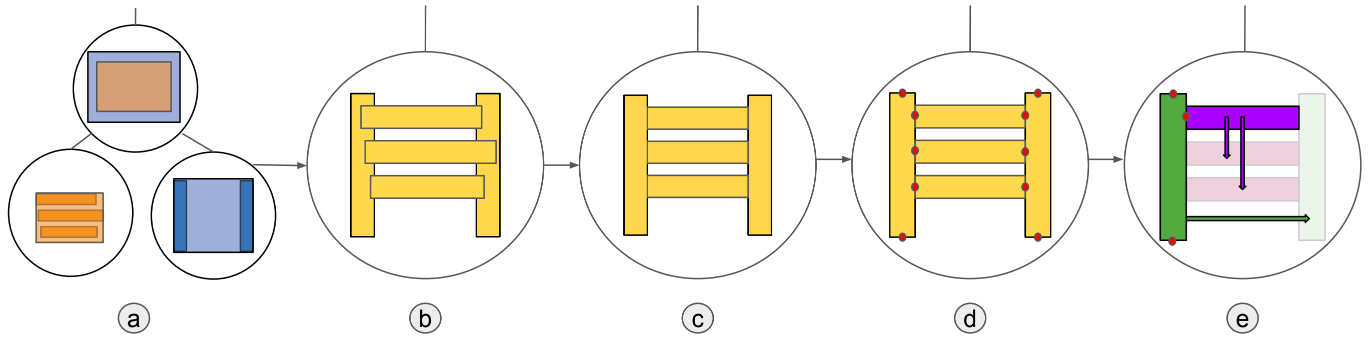

Fig. 5. The steps of our program extraction pipeline. *(a)* Fragment of an input hierarchical part graph showing chair back (parent node), chair back frame (blue child), and chair back surface (orange child). *(b)* Locally flattening the hierarchy so that physically interacting leaf parts become siblings. *(c)* Shortening leaf parts that intersect other leaf parts. *(d)* Locating attachment points between parts. *(e)* Forming leaf parts into symmetry groups.

satisfy the attachment (see Appendix A for details). Higher-level macros expand into two or more `Cuboid` or `attach` lines, which are then immediately executed (see Appendix B for details).

One distinct advantage of this imperative semantics, as opposed to an alternative formulation in which the program specifies constraints which are jointly optimized, is that the entire process of executing a program is end-to-end differentiable. That is, it is possible to compute the gradient of the program's output geometry with respect to the continuous parameters in the text of the program (e.g., cuboid dimensions, attachment point locations). We make use of this feature in results shown later in this paper.

Handling hierarchy. Thus far, we have described a language that can generate flat assemblies of parts, but not hierarchical ones. The extension to hierarchical shapes is straightforward: we represent hierarchical shapes by treating select non-leaf cuboids as the bounding box of another program (e.g., the contents of its "BBlock"). Figure 3 shows an example of a program in which cuboids expand into subprograms.

5 TURNING SHAPES INTO TRAINING PROGRAMS

ShapeAssembly allows us to write programs that generate new shapes. However, we are interested in using the language to represent existing shapes in a dataset, so that we can learn to generate novel instances from the same underlying shape distribution. In this section, we describe how we accomplish this goal. Given an input shape, represented as a hierarchical part graph, the process divides into three steps: extracting program information, creating candidate programs, and checking program validity.

5.1 Extracting Program Information

To convert hierarchical part graphs into ShapeAssembly programs, we perform a series of data regularizations, record cuboid parameters, locate cuboid-to-cuboid attachments, and identify symmetry groups (Figure 5). We provide a high-level overview of the steps involved here, and a detailed description in Appendix C.

Regularization. Before we parse program attributes, we attempt to create more regularized part graphs through a series of data-cleaning steps. For instance, in the flattening phase, we restructure the part graph hierarchy so that leaf parts with spatial relationships are more often siblings. In the shortening phase, we decrease the dimensions of leaf cuboids that interpenetrate other leaf cuboids (to create more surface-to-surface part connections).

Cuboids. Ground truth cuboid dimensions are provided in the input part graphs. A cuboid is marked as aligned if its orientation matches its parent cuboid (with an allowable error of 5-degrees).

Attachment. To locate cuboid-to-cuboid attachments, we sample a uniform, dense point cloud on each cuboid in the scene. For each pair of cuboids, we compute the intersection of the point clouds. If the intersection set is non-zero, we record an attachment point within the volume formed by the intersection, with preference for locations on the centers of faces. For every cuboid, we then check if any of its parsed attachments could be represented as a squeeze relationship, and replace any that can.

Symmetry. To find symmetry groups, we identify collections of cuboids that share a reflectional or translational relationship about either the X, Y, or Z axis of their parent cuboid. For each collection, if all of the member cuboids have the same connectivity relationships, we form them into a symmetry group. Each symmetry group is represented by a transform applied to a single cuboid, and all other members are removed from the graph.

5.2 Creating Candidate Programs

Given the extracted program information, we know the content of the program, but not how the lines should be ordered. To make the task of learning a generative model of programs easier, we aim to extract only a single, "canonical" program for each shape. As the ordering of cuboid and symmetry lines doesn't change the executed geometry, this consistency is enforced by ordering these lines according to the semantic label of each part involved in the line. Ties in this ordering between same part-type cuboids are broken by sorting on centroid position.

Deciding on a single ordering of the attach and squeeze statements is more challenging. Since ShapeAssembly has an imperative execution semantics, the order in which these commands are executed is significant: different orderings can potentially create different output geometries. To reduce the space of possible orderings, we

only consider programs which follow a *grounded* attachment order, which we define as follows:

- Initially, only the shape bounding box is grounded.
- The only valid attachments to perform are those which connect a cuboid to a grounded cuboid.
- After executing an attachment, the newly-attached cuboid becomes grounded.

If there are multiple valid grounding orders, we first discard any orderings that produce worse geometric fits to the target shape. If ambiguities in the attachment ordering still remain, we break ties using (1) the semantic ordering of the cuboids involved in the attachment (2) preferring attachments from non-aligned to aligned cuboids and finally (3) preferring attachments from cuboid face-centers.

5.3 Validating Programs

Once we extract a canonical ShapeAssembly program, we perform a series of checks to verify the results of our procedure. Programs must pass the following validation steps in order to be added to our training data:

Reconstruction. Executed programs should recreate the geometry of their respective ground truth part graph. To verify this, we sample point clouds from the surfaces of the ground truth shape and the geometry generated by executing the canonical program. These point clouds are compared using the F-score [Knapitsch et al. 2017] metric; a program is filtered out if it produces an F-score lower than 75.

Semantics. Programs must respect the semantics of ShapeAssembly. For instance, within each program, the connectivity graph of all parts should have only one component. Likewise, executed programs should not create geometry that extends beyond the bounding volumes they define.

Complexity. Programs that are overly complex (more than 12 leaf cuboid instantiations) are discarded. Note that, when executed, programs can still produce more than 12 leaf cuboids through expanding symmetry macros.

6 LEARNING TO GENERATE PROGRAMS

Given the programs extracted from our dataset, we now have the data we need to train a neural network to write novel hierarchical ShapeAssembly programs for us. In this section, we describe the generative model architecture we use, our learning procedure, and how we sample new shapes from the learned model.

6.1 Model Architecture

Figure 6 shows our generative model architecture. It is a hierarchical sequence VAE. The encoder branch embeds a hierarchical ShapeAssembly program into a latent space. The decoder branch converts a point in this latent space into a hierarchical ShapeAssembly program. The bottleneck of our network is parameterized by separate μ and σ vectors in the standard variational autoencoder (VAE) setup.

The dark grey callout in Figure 6 illustrates the operation of our decoder within a single node of the program hierarchy. The decoder receives as input the latent code z_{par} of its parent node (or the root latent code from the encoder, if it is the root node of the hierarchy). This latent code is used to initialize the hidden state of a Gated Recurrent Unit (GRU), a recurrent language model which is responsible for constructing a representation of the program state. The output of the GRU cell is sent to the line decoder sub-routine, which predicts a line in the ShapeAssembly grammar, that is then passed as input back to the GRU cell at the next time step.

The purple callout in Figure 6 gives a detailed depiction of the line decoder sub-routine. The line decoder receives the hidden state of the GRU cell, along with conditioning information about the size of the current bounding volume, and uses a collection of multilayer perceptrons (MLPs) to predict a 63-dimensional vector representing a single line in ShapeAssembly . The sub-networks it uses are:

- $f_{\textbf{cmd}}$**: (7):** Predicts the type of command to execute. This is a one-hot vector whose seven entries correspond to `<start>` (the special program start token), `<stop>` (the special program stop token), `Cuboid`, `attach`, `squeeze`, `translate` and `reflect`.
- $f_{\textbf{cube}}$**: (4):** Predicts the length, width, height, and aligned flag for `cuboid` lines, conditioned on the bounding volume dimensions.
- $f_{\textbf{idx}}$**: (11 × 3):** Predicts the indices of the cuboids involved in the line represented as 3 one-hot vectors, conditioned on the predicted command. We limit each node in the hierarchy to contain at most 10 children parts, so there are 11 choices (10 cuboids and the bounding volume).
- $f_{\textbf{att}}$**: (3 × 2):** Predicts the (x, y, z) coordinates involved in an `attach` line, conditioned on the cuboids involved in the attach.
- $f_{\textbf{sqz}}$**: (8):** Predicts the the face involved in a `squeeze` line as a one-hot vector in the first 6 indices. The last 2 indices predict the (u, v) coordinates. Both predictions are conditioned on the cuboids involved in the squeeze operation.
- $f_{\textbf{sym}}$**: (5):** Predicts the axis involved in a symmetry line as a one-hot vector in the first 3 indices. For `translate` lines, the 4th index is the number of cuboids involved in the symmetry group, and the 5th index is the scale of the symmetry. All predictions are conditioned on the cuboid involved in the symmetry and the bounding volume dimensions.

Hierarchical decoding. To generate a hierarchical program, our decoder also includes a submodule f_{child} which is executed after every predicted `Cuboid` command to determine whether that cuboid should be recursively expanded. This is another MLP which takes as input both the current hidden state of the GRU as well as z_{par}, the overall latent code for this hierarchy node. f_{child} produces two outputs: a Boolean flag for whether the current cuboid should be expanded into a child program, and a new latent code z_{child} which is used to initialize the decoder for this child program.

6.2 Learning Procedure

We implement our models in PyTorch[Paszke et al. 2017]. All training is done with the Adam optimizer [Kingma and Ba 2014], with a learning rate of 0.0001 without batching. All multilayer perceptrons have 3 layers and use leaky ReLU [Maas 2013] with $\alpha = 0.2$.

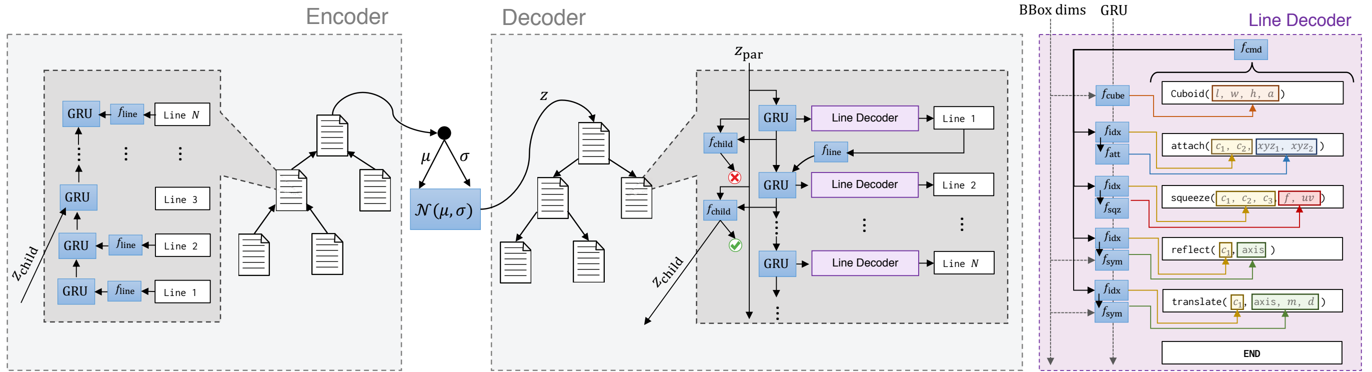

Fig. 6. Architecture of our hierarchical sequence VAE for ShapeAssembly programs. Given a ShapeAssembly program, the encoder ascends the hierarchy from the leaves to the root, encoding each sub-program into a latent z vector. Given a latent code, the decoder recursively decodes a hierarchical ShapeAssembly program. Within each hierarchy node, a recurrent neural network decodes each line of the program.

We train our model in a seq2seq fashion, where the ground truth input sequence is teacher forced to the model, and our model is tasked with predicting each subsequent line. During training, we use a program reconstruction loss that only considers entries of the predicted 63 dimensional vector that are relevant to the target line. For instance, when predicting a `Cuboid` line, no part of the reconstruction loss comes from the indices in the tensor associated with symmetry. The program reconstruction loss is comprised of a cross-entropy component for each one-hot prediction (with weight 1) and an l1 loss for each continuous component (with weight 50). Additionally we use a KL loss in the standard VAE setup with weight 0.1 [Kingma and Welling 2014].

Enforcing semantically-valid output. As our model generates shape *programs*, rather than raw shape geometry, we can use the semantics of the ShapeAssembly language to detect outputs that would be invalid, and prevent them from happening. For instance, attaches must be made in a grounded order. If a predicted `attach` line violates such a constraint, we use a backtracking procedure to find new 'valid' parameter values whenever possible. During unconditional generation, if we cannot fix the line through backtracking, we reject the sample. During interpolation, if we cannot fix the line through backtracking we don't add the predicted line to the program. Appendix D describes the complete semantic validity procedure we enforce. We also note that this approach to forbidding the generation of invalid outputs is similar to that of the Grammar Variational Autoencoder [Kusner et al. 2017]. However, that model only uses grammar *syntax* to determine whether an output is valid, whereas as we use program *semantics*.

7 RESULTS AND EVALUATION

In this section, we demonstrate our learned generative model's ability to synthesize high-quality hierarchical ShapeAssembly programs, and we compare it to alternative generative models of 3D shape structure. All of the experiments described were run on a GeForce RTX 2080 Ti GPU with an Intel i9-9900K CPU, and consumed 3GB of GPU memory.

We use objects from the PartNet dataset [Mo et al. 2019b] as our training data. It contains 3D shapes in multiple categories, each with a hierarchical part segmentation and labeling. For the experiments in this paper, we use the *Chairs*, *Tables*, and *Storage* categories. After running the program extraction procedure described in Section 5, we obtain 3835 ground truth programs from *Chairs*, 6536 ground truth programs from *Tables*, and 1551 ground truth programs from *Storage*.

7.1 Novel Shape Synthesis

In this section, we present both qualitative and quantitative evaluations of our method's ability to produce novel shape structures. Figure 7 includes some unconditionally generated samples from our learned generative model for each of the three shape categories. Above each sample we show its nearest neighbor in the training data based on Chamfer distance. Additionally, below each sample we visualize its nearest neighbor in the training data based on program distance, the string edit distance of a tokenized version of our hierarchical programs. As shown, our method is able to generate complex and interesting structural variation without copying either the geometry or program structure of its training data.

As our model directly generates programs, its outputs can be easily edited to produce variants. In Figure 8 we demonstrate how by changing just the continuous parameters of programs generated by our model, we are able to create a wide variety of output geometry, all the while maintaining part-to-part attachment relationships.

We compare the generated results of our method against two baselines:

- **StructureNet** is a variational autoencoder that generates hierarchical part graphs with cuboids at each node [Mo et al. 2019a].
- **3D-PRNN** is a recurrent neural network that generates a sequence of cuboids [Zou et al. 2017]. It enforces global bilateral symmetry by only generating cuboids with some part of their geometry on the negative side of the $x = 0$ plane, and then reflecting generated cuboids which fall entirely on that side of the plane.

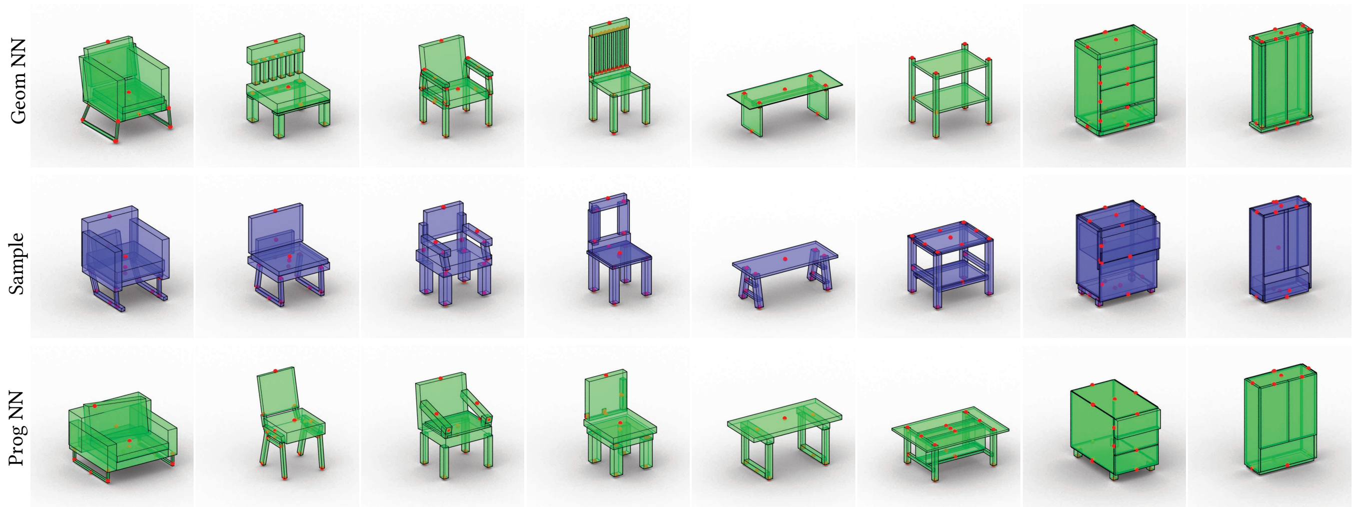

Fig. 7. In the middle row, we show samples from our generative model of ShapeAssembly programs. In the top row, we show the nearest neighbor shape in the training set by Chamfer distance. In the bottom row, we show the nearest neighbor shape in the training set by program edit distance. Our method synthesizes interesting and high-quality structures that go beyond direct structural or geometric memorization. We quantitatively examine ShapeAssembly's generalization in Table 4. Refer to the supplemental material for the corresponding program text.

We compare against the StructureNet models released by the authors. These were trained on the subset of PartNet that they were able to represent within the constraints of their problem formulation. This is a heavily overlapping set, but not identical, with the shapes we were able to find valid ShapeAssembly programs for. In direct comparisons with StructureNet for reconstruction tasks, we only consider shapes that appear in the validation splits of both methods. We compare against a version of 3D-PRNN that was re-trained on the data we use for our generative model. Figure 9 shows a qualitative comparison of unconditionally generated samples from each method. Our method is capable of generating diverse, structurally complex, 3D shape structures across multiple categories. Attachment as a primary operation provides a strong inductive bias for generating physically plausible shapes that maintain realistic part-to-part relationships. In contrast, both comparison methods that directly predict part placements in 3D space are prone to producing floating cuboids or jumbled collections of spatially colocated parts.

7.1.1 Analysis of Shape Quality. We also quantitatively compare the quality of the shape structures generated by different methods. Our desiderata for generated shape structures is that they should be physically plausible and come from the same distribution that the model was trained on. In order to asses the quality of generated output, we use the following metrics:

- **Rootedness ⇑ (% rooted):** The percentage of shapes for which a connected path exists between the ground and all leaf parts.
- **Stability ⇑ (% stable):** The percentage of shapes which remain upright under gravity and small forces in a physical simulation.
- **Realism ⇑ (% fool):** The percentage of test set shapes classified as "generated" by a PointNet classifier trained to distinguish between generated shapes and shapes from the training dataset.
- **Frechet Distance ⇓ (FD):** Measurement of distributional similarity between generated shapes and the training dataset using the feature space of a pre-trained PointNet model [Heusel et al. 2017]

Further details about these metrics are provided in Appendix E.

We show results for these metrics on 1000 unconditional generated shapes in Table 2. Our method largely outperforms 3D-PRNN and StructureNet across these metrics for three categories of shapes. While StructureNet achieves good rootedness scores, especially

Fig. 8. Programs, by way of representational form, allow for easy semantic editing of generated output. Each column shows a sample from our model in the top row. In the bottom row we create a variant with the same structure, but different geometry, by editing only the continuous parameters of the program. Program text can be found in the supplemental material.

for the Storage category, our method performs better in the other three metrics along all categories. The samples from 3D-PRNN, achieve similar FD and % fool scores with StructureNet, but perform markedly worse on the rootedness and stability metrics.

Additionally in this experiment we compare our model with a series of ablated versions:

- **Flat:** Training on programs with no hierarchies, only leaf parts.
- **No Order:** Training on programs without canonical ordering as described in Section 5.
- **No Align:** Training on programs without an aligned flag for cuboids.
- **No Macros:** Training on programs without `squeeze`, `translate`, or `reflect` commands.
- **No Reject:** At generation time, discard unfixable, invalid program line predictions instead of rejecting the entire sample.

Training without hierarchy (Flat) slightly improves rootedness, but drastically lowers the quality of output as seen in the % fool and FD columns. Training on programs without a canonical ordering (No Order) performs worse on every metric. Removing the alignment flag (No Align) actually improves performance on the Chair category for % rooted and % fool, but drastically worsens the physical validity of generations for Tables and Storage, categories where parts are much more often aligned with their parent cuboid. Training without macros (No Macros) once again decreases the performance of all metrics, but not by a substantial margin. Finally, we see that while the rejection sampling step does improve the quality of our generated samples, without it we still outperform 3D-PRNN and StructureNet by a wide margin.

Table 2. Comparing the quality of generated samples. Our method outperforms other generative methods for 3D shape structure in terms of realism and physical validity. Through a series of ablation baselines, we validate various design decisions of our method.

Category	Method	% rooted ⇑	% stable ⇑	% fool ⇑	FD ⇓
Chair	3D-PRNN	73.1	50.9	12.60	39.30
	StructureNet	89.7	74.9	4.04	64.79
	Ours (Flat)	**95.0**	60.0	11.58	77.45
	Ours (No Order)	82.4	58.4	12.36	64.17
	Ours (No Align)	94.6	84.6	**28.68**	29.32
	Ours (No Macros)	92.0	77.9	19.56	36.78
	Ours (No Reject)	92.9	79.7	23.36	**20.63**
	Ours	94.5	**84.7**	25.06	22.34
	Ground Truth	100	88.0	—	—
Table	3D-PRNN	71.2	29.4	2.12	140.07
	StructureNet	94.4	76.8	3.94	173.35
	Ours (Flat)	87.0	66.0	29.84	148.63
	Ours (No Order)	84.5	56.0	27.38	114.10
	Ours (No Align)	92.2	61.5	23.64	**46.64**
	Ours (No Macros)	95.9	85.0	33.16	53.21
	Ours (No Reject)	94.1	76.4	29.20	52.78
	Ours	**96.2**	**85.9**	**33.21**	49.07
	Ground Truth	100	93.1	—	—
Storage	3D-PRNN	44.8	20.8	4.62	94.08
	StructureNet	**96.2**	75.0	5.04	92.85
	Ours (Flat)	95.9	74.0	7.44	81.17
	Ours (No Order)	87.9	63.4	8.70	107.42
	Ours (No Align)	89.7	49.3	11.04	**30.15**
	Ours (No Macros)	87.5	69.9	5.92	72.80
	Ours (No Reject)	94.3	80.9	11.66	31.69
	Ours	95.3	**83.7**	**13.50**	31.72
	Ground Truth	100	87	—	—

7.1.2 Analysis of Editability. In this section, we quantitatively analyze our previous claim that directly predicting programs improves editability. We claim that a program is more editable if it is both compact and compromised of higher level functions. That is, a shorter program that uses higher-level constructs will be easier to understand and make changes to.

As a strong baseline, we evaluate the editability of our programs against the generated outputs of 3D-PRNN and StructureNet. As 3D-PRNN and StructureNet do not directly produce ShapeAssembly programs, we use our extraction procedure described in Section 5 in order to convert their generations into programs. As StructureNet predicts part graph hierarchies, the representational form our extraction procedure takes as input, we use our procedure without any of the data cleaning steps. As 3D-PRNN has no notion of hierarchy, we create single node part graphs out of their output samples, which are then run through our program extraction logic.

Table 3 shows results from an experiment where we compare the ShapeAssembly programs of each method's generations (directly predicted by our method, parsed programs from comparisons). The metrics we use are the number of lines in each program (as a coarse measure of compactness) and the percentage of lines which are macros (split by macro type).

Compared with programs parsed from StructureNet, the programs generated by our model are much more compact and have higher rates of macro usage across all categories of shapes. While our method also has higher macro rate usage compared with 3D-PRNN, 3D-PRNN programs are more compact in the Chair and Table categories. Based on 3D-PRNN's poor performance within our shape quality experiments (Table 2), and its significant deviation from the number of lines found in the ground truth programs (the cleanest set of ShapeAssembly programs we have access to), there is reason to believe that the compactness of its parsed programs more likely reflects shape simplicity rather than useful editability.

7.1.3 Analysis of Variability. Beyond quality and editability, we also consider the variability of outputs of each method. Specifically, for generated shapes, we care about their novelty with respect to the training data, their complexity, and their variety. We present results of an experiment using Chamfer distance to quantify performance across these areas in Table 4.

The *Generalization* metric measures the average distance of each generated sample to its nearest neighbor in the training set. As all methods have higher generalization scores than the validation set, we can conclude that none of the methods appear to be overfitting. For our method specifically, this re-enforces the qualitative nearest neighbor results presented in Figure 7.

The *Coverage* metric measures the average distance of each validation shape to its nearest neighbor in the set of generated shapes.

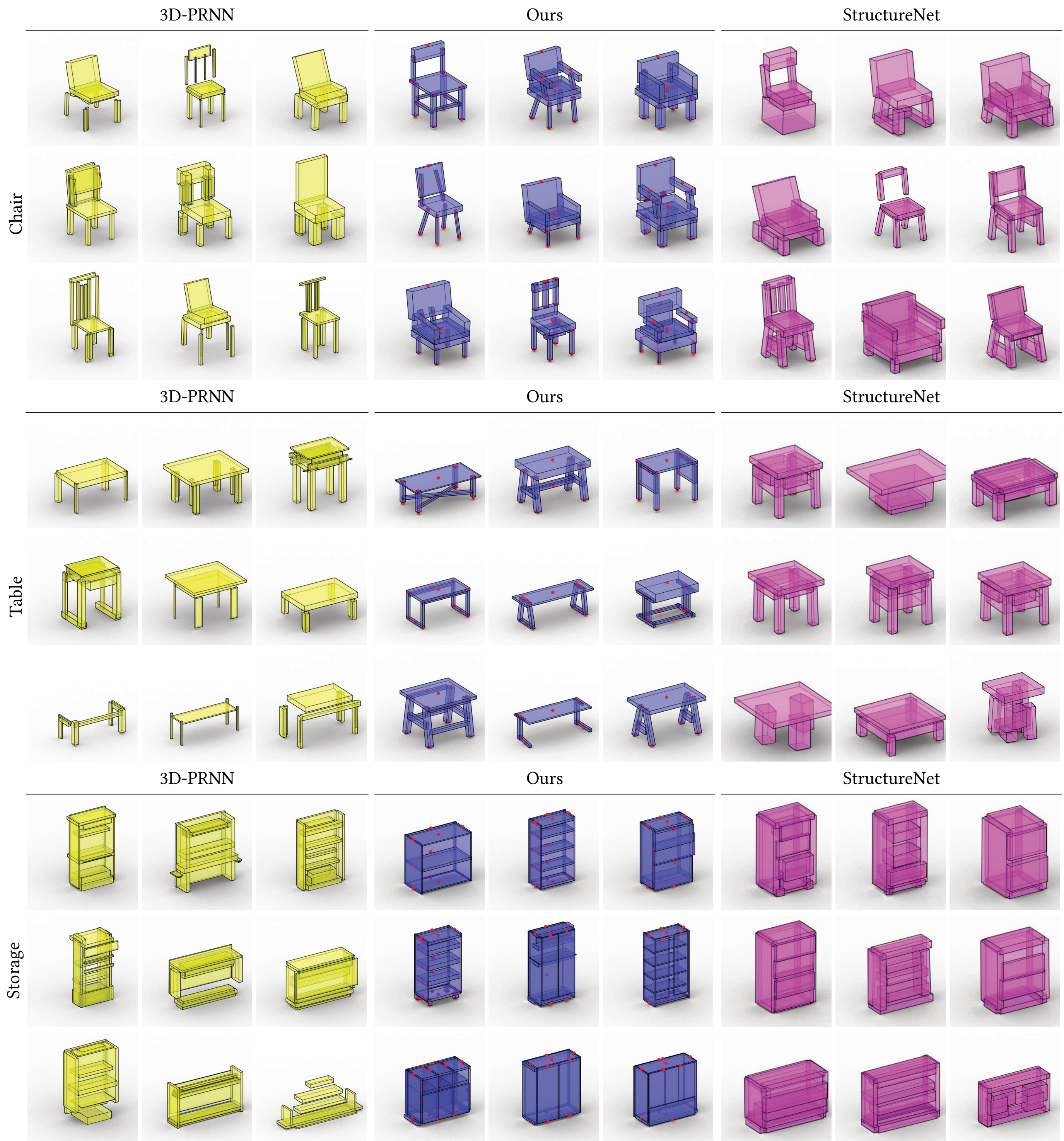

Fig. 9. Qualitative comparison between generated samples from our method, StructureNet, and 3D-PRNN. Across different categories, our method creates novel ShapeAssembly programs that, when executed, produce shape structures that maintain realistic and physically valid part-to-part relationships. Comparison methods that directly predict 3D shape geometry exhibit failure cases where parts become disconnected or intersect in an implausible manner.

Table 3. Markers of program editability for ShapeAssembly programs predicted by our generative model compared with ShapeAssembly programs parsed from outputs of other generative methods. Training our model in the space of programs allows us to represent geometry more compactly. We find higher rates of macro functions per program line in our method's generations compared with extracting programs from other generative models' predictions.

Category	Method	Lines ⇓	Macros Per Line: Refl ⇑	Trans ⇑	Squeeze ⇑	Total ⇑
Chair	3D-PRNN	**15.7**	**0.1100**	0.0020	0.0240	0.1430
	StructureNet	27.1	0.0600	0.0004	0.0700	0.1330
	Ours	20.4	0.0880	**0.0054**	**0.0920**	**0.1860**
	Ground Truth	24.4	0.0800	0.0090	0.1130	0.2070
Table	3D-PRNN	**13.1**	**0.1300**	**0.0010**	0.0680	0.1990
	StructureNet	24.8	0.0270	0.0006	0.0620	0.0900
	Ours	19.0	0.0990	0.0002	**0.1440**	**0.2440**
	Ground Truth	20.0	0.0950	0.0050	0.1450	0.2460
Storage	3D-PRNN	22.6	0.0170	0.0060	0.0530	0.0770
	StructureNet	30.7	0.0390	0.0040	0.0770	0.1200
	Ours	**19.8**	**0.0820**	**0.0080**	**0.1440**	**0.2340**
	Ground Truth	24.7	0.0650	0.0147	0.1510	0.2320

Across all categories our method achieves the best results, and by a wide-margin for tables, which indicates that our generations have enough complexity to match the distribution of the validation shapes.

The *Variety* metric measures the average distance of each generated shape to its nearest neighbor in the set of generated shapes besides itself. Once again, across all categories our method achieves top, or tied for top performance.

Additionally, we look at average number of leaf parts as a coarse proxy for the complexity of a shape's structure, which is shown in Table 5. While our method has a similar number of leaf parts to the comparison methods for the Chair and Table categories, we do have fewer leaf parts on average for Storage. Qualitatively, these additional parts in the comparison methods often manifest as collections of spatially colocated cuboids, and not necessarily more complex shape structures.

In terms of the variability of programs generated by our method, we note that 65% of Chair programs, 85% of Table programs, and 53% of Storage programs contained ShapeAssembly program structures not present in the training data. Thus our method not only exhibits novelty in the geometric domain, but also in the structural domain.

7.1.4 Program Clustering. Our approach is predicated on the assumption that a single program can represent a parametric family of multiple shapes, allowing for this shape space to be explored via manipulation of interpretable program parameters. To verify whether this is true, we cluster shapes that are represented by *structurally-equivalent* programs (i.e. programs that are the same up to continuous parameter variations). Figure 10 shows program clustering results for the ground truth programs we parse from PartNet. These results demonstrate how the structure of a single ShapeAssembly program is able to represent related shapes through different parameterizations. The marked improvement in clustering when

Table 4. We compare the geometric variability of generated shapes from different methods. In the first column, we measure generalization as the average nearest neighbor distance (NND) from generated samples to shapes in the training set. In the second column we measure coverage as the average NND from shapes in the validation set to generated samples. In the last column, we measure variety as the average NND from shapes in the generated samples to other generated shapes in the same set. Across three categories of shapes, our method performs the best on the coverage and variety metrics, while outperforming validation on generalization (demonstrating we are not overfitting).

Category	Method	*Generalization* NND to Train ⇑ CD	*Coverage* NND from Val ⇓ CD	*Variety* NND to Self ⇑ CD
Chair	3D-PRNN	**0.111**	0.123	**0.104**
	StructureNet	0.104	0.119	0.087
	Ours	0.108	**0.118**	**0.104**
	Validation	0.105	—	0.114
Table	3D-PRNN	0.095	0.130	0.086
	StructureNet	**0.129**	0.141	0.0925
	Ours	0.101	**0.108**	**0.102**
	Validation	0.09	—	0.099
Storage	3D-PRNN	**0.134**	0.132	**0.119**
	StructureNet	0.129	0.135	0.107
	Ours	0.125	**0.129**	**0.119**
	Validation	0.11	—	0.125

Table 5. We compare the average number of leaf parts in generated shapes, as a coarse proxy for complexity of shape structure. Our method generates similar numbers of leaf parts compared with other methods for Chairs and Tables, but fewer leaf parts for Storage. Qualitatively, the additional leaf parts measured in comparison methods often manifests as spurious overlapping cuboids, rather than more complex structural variety.

Category	Method	Avg # Leaf Parts
Chair	3D-PRNN	8.6
	StructureNet	8.7
	Ours	7.9
	Ground Truth	9.7
Table	3D-PRNN	7.07
	StructureNet	8.16
	Ours	7.84
	Ground Truth	8.4
Storage	3D-PRNN	10.6
	StructureNet	12.3
	Ours	8.4
	Ground Truth	10.8

splitting by intermediate part programs compared with clustering on entire shape programs, provides additional support for our hierarchical approach; shape programs are more likely to share structure within a node of the hierarchy than they are to match entire hierarchies exactly.

7.1.5 Synthesizing Surface Geometry. While collections of part proxies are a useful modeling representation for 3D shape structures, they do not directly attempt to capture the wide range of intra-part variability present in man-made objects. We demonstrate how ShapeAssembly programs can additionally be used to model

Table 6. We measure smoothness along random high-frequency interpolation sequences in each method's latent space. The Geo column measures smoothness with Chamfer distance, while the Prog column measures smoothness with program edit distance. Note that 3D-PRNN is missing because it is not a latent variable model and thus does not support interpolation.

Category	Method	Avg. Step Size ⇓ Geo	Prog
Chair	StructureNet	**0.0384**	3.90
	Ours	**0.0384**	**1.33**
Table	StructureNet	0.0474	4.75
	Ours	**0.0389**	**2.48**
Storage	StructureNet	0.0512	4.29
	Ours	**0.0482**	**2.6**

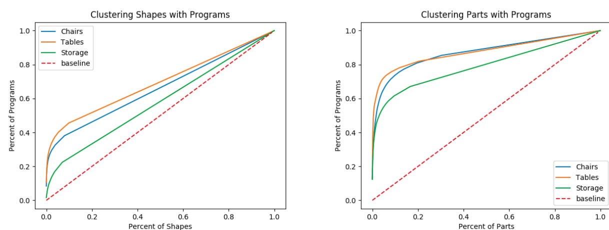

Fig. 10. Clustering results that demonstrate how the structure of a single ShapeAssembly program is capable of capturing a family of related shapes. Using ground truth programs found with our program extraction procedure, in the left graph we plot the percentage of shapes captured as we consider more program structures extracted from the data. In the right graph we show the same plot but with parts (nodes) instead of shapes (full hierarchy).

parts at finer levels of detail by turning ShapeAssembly programs into dense point clouds. As a proof of concept, we augment our generative model with a point cloud encoder that consumes dense point cloud samples of ground truth leaf parts, and a point cloud decoder that generates dense point clouds for every leaf part within its predicted bounding volume. Figure 11 shows some qualitative results of our method, trained on point clouds sampled from the dense geometry of Chairs found in PartNet. These generated surfaces provide additional detail over the geometry specified by their cuboid part proxies, as evidenced by both the rounding in the legs and back slats, and also in the curvature of the chair back surfaces.

7.2 Latent Space Interpolation

Beyond novel shape generation, we evaluate the ability of our method to interpolate between two points in our latent space. The presence of smooth, semantic transitions between end-points indicates a well-formed latent space. In Figure 12 we qualitatively compare our method with StructureNet on the task of interpolating between shapes in the validation sets of both models. Our interpolations demonstrate both geometrically smooth and semantically consistent transitions. For instance, in the top interpolation sequence,

Fig. 11. Converting generated ShapeAssemblyprograms into dense point clouds. We use a point cloud decoder to predict the surface geometry of each leaf part proxy in our 3D shape structure. In this process, geometric details begin to take form, at the cost of some artifacts. We discuss a method for improving this procedure in section 8.

the surface of the chair back in the source shape gradually shrinks vertically until in the target shape it is just a horizontal bar. At the same time, the number of vertical slats in the chair back gradually increases from 2, to 4, to 5.

In Table 6, we attempt to quantify the smoothness along random interpolation sequences within the latent space of each generative model. In this experiment, 100 interpolation sequences were computed from sources to targets that were randomly sampled in each model's latent space, with 100 interpolation steps per sequence. Each method's geometric smoothness is computed by taking the average Chamfer distance (normalized by shape scale) between each interpolation step. The lower geometric smoothness of our method, compared to StructureNet in the Table and Storage categories, demonstrates the quality of the latent space learned by our method. Moreover, using our procedure to turn StructureNet outputs into ShapeAssembly programs, we can measure the program smoothness along these interpolation paths. Each method's program smoothness is computed by taking the average tokenized program edit distance between each interpolation step. As a measure for structural change throughout the transitions of an interpolation sequence, our lower program smoothness metric again shows how our method benefits by operating within the space of 3D shape programs.

7.3 Synthesis from Unstructured Geometry

Another way to inspect the structure of a generative model's latent space is through performing "synthesis from X", by projecting X into the latent space of the generative model. As an application for 3D reconstruction, we are able perform such a projection with point clouds, demonstrating how our generative model's latent space can synthesize ShapeAssembly programs from unstructured geometry.

Specifically, we train a PointNet++ encoder [Qi et al. 2017] to map point clouds sampled on dense mesh geometry to the latent space learned by our generative model. These latent codes are then converted into programs by our trained decoder.

In Table 7, we show an experiment comparing our method against StructureNet on the task of reconstructing point cloud samplings

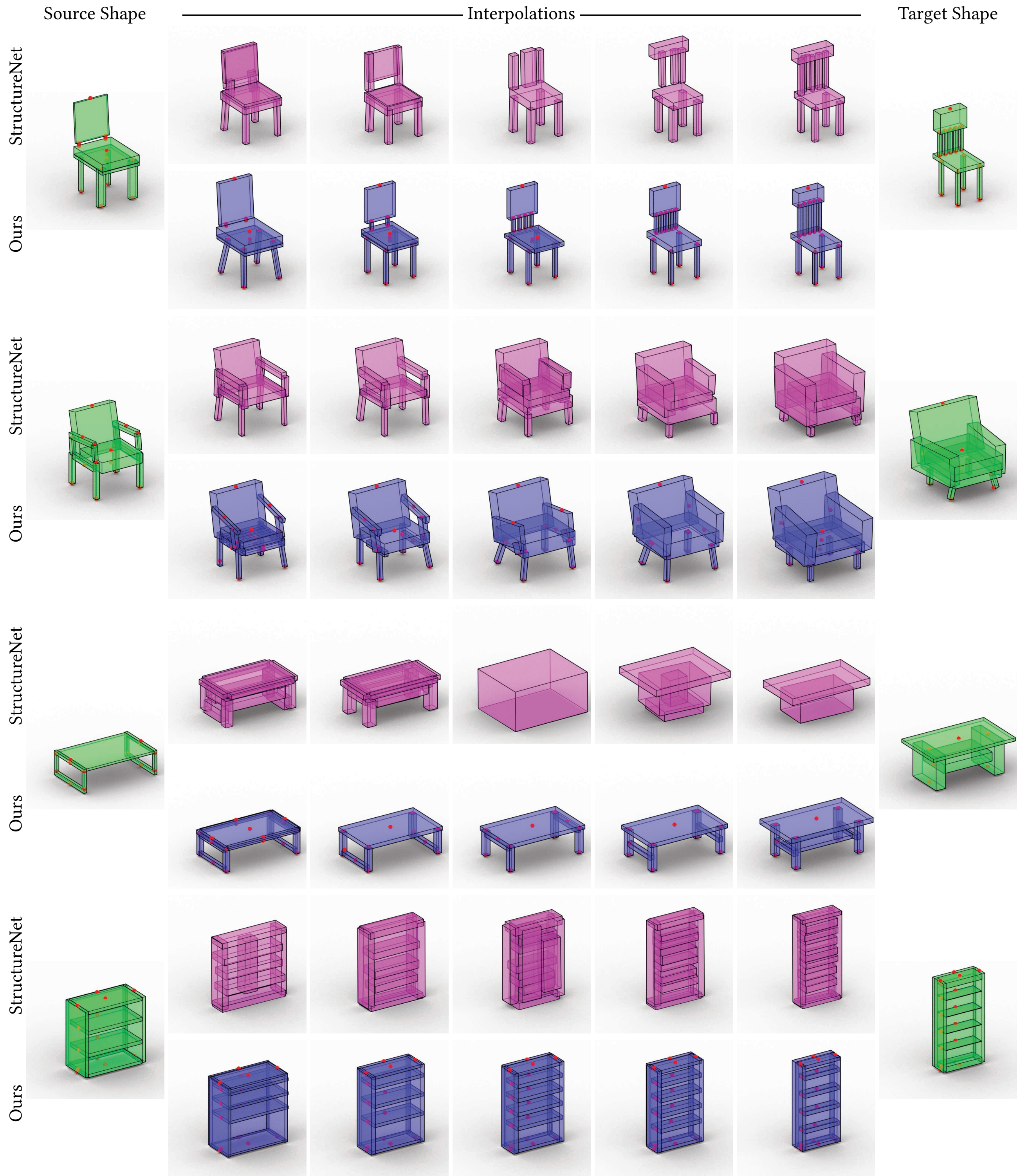

Fig. 12. A qualitative comparison of latent space interpolation between our method and StructureNet on shapes from the validation set. Our method's interpolations within program space produce sequences that combine smooth continuous variation with discrete structural transitions.

Table 7. Results from our point cloud reconstruction experiment. Our model's well-formed latent space allows for more accurate and physically valid reconstructions without further optimization. With additional optimization, using the reconstructed program from our method and our differentiable interpreter finds the best trade-off between reconstruction accuracy and maintaining physical validity.

Method	F1 ⇑	% rooted ⇑	% stable ⇑
StructureNet	24.3	95.1	78.4
Ours	**31.1**	**95.5**	**84.4**
SN + Opt Cuboids	**80.0**	92.9	72.7
SN + Opt Program	77.4	90.0	71.9
Ours + Opt Cuboids	77.6	93.1	72.9
Ours + Opt Program	75.8	**95.3**	**80.2**

of dense geometry on the intersection of each method's validation set for Chairs in Partnet (463 shapes total). We evaluate reconstruction accuracy with F-score [Knapitsch et al. 2017], and the physical validity of reconstructions with the rootedness and stability metrics. When projecting point clouds into the latent space of each method (top two rows), our method outperforms StructureNet on both reconstruction accuracy and maintaining physical validity. This demonstrates, once again, the well-structured nature of our method's latent space.

Moreover, as the ShapeAssembly interpreter is differentiable, we can further refine the continuous parameters of a program by minimizing the Chamfer distance between executed geometry and a target point cloud with a gradient-based optimizer. We compare this procedure **(Ours + Opt Program)** against the following conditions:

- **SN + Opt Cuboids**: Starting with StructureNet's reconstruction, then directly optimizing predicted cuboids to minimize Chamfer distance to the target point cloud.
- **SN + Opt Program**: Parsing StructureNet's reconstruction into a ShapeAssembly program, then optimizing the program to minimize Chamfer distance to the target point cloud.
- **Ours + Opt Cuboids**: Starting with our reconstruction, directly optimizing predicted cuboids to minimize Chamfer distance to the target point cloud.

We show results for this experiment in the last four rows of Table 7. All of the optimization procedures improve reconstruction accuracy at the cost of physical validity. However, Ours + Opt Program is the only condition that achieves a desirable trade-off in this exchange, gaining much more reconstruction accuracy improvement than it loses in physical validity.

We show some qualitative results of this experiment in Figure 13. Through latent space projection, our model is able to output the rough 3D shape structure (column 1) of an input unstructured point cloud (column 0). Through our differentiable interpreter, we are able to find continuous parameters for the predicted program structure that ultimately lead to better reconstruction fits (column 3). Shape programs place a strong structural regularization prior over unstructured 3D data, and thus our presented method is less prone to "losing" semantic parts, such as small legs, in comparison to the other conditions.

8 CONCLUSION AND FUTURE WORK

In this paper, we took a first step toward marrying the complementary strengths of neural and procedural 3D shape generative models by introducing a hybrid neural-procedural approach for synthesizing novel 3D shape structures. We introduced ShapeAssembly, a low-level "assembly language" for shape structures, in which shapes are constructed by declaring cuboidal parts and attaching them to one another. We also introduced a differentiable interpreter for ShapeAssembly, allowing the optimization of program parameters to produce desired output geometry. After describing how to extract consistent programs from existing shape structures in the PartNet dataset, we then defined a deep generative model for ShapeAssembly programs, effectively training a neural network to write novel shape programs for us. We evaluated the quality of the generative model along several axes, showing that it produces more plausible and physically-valid shapes, and that its latent space is better-structured than that of other generative models of shape structure. We also found that directly generating shape programs leads to more compact, editable programs than extracting programs from shapes generated by methods that directly output 3D geometry.

Limitations. As mention in Section 5, we do not successfully extract training programs from every shape in our dataset. For instance, our program extraction procedure assumes that the orientation of all parts can be specified through solely part-to-part attachments, yet as demonstrated in Figure 14, this does not hold for all shapes. While it is possible to reconstruct these shapes with ShapeAssembly programs (through attaching parts to "floating" points in space via the bounding volume) such programs will never be added to our training data, and thus our generative model won't learn to produce such constructs. Our design decision to discard training programs with more than 12 total `Cuboid` declarations has a similar effect: it limits our generative model from synthesizing the most complex of shape structures that exist in our dataset. We impose such strict criteria in order to make our training programs exhibit more regularity, simplifying the learning task for our neural network at the expense of its potential expressivity.

This highlights a central tradeoff: higher variability in the training programs may result in lower quality shapes synthesized by a generative model. This phenomenon is not unique to our setting: it is well-known that e.g., image generative models perform better on very-regularly-structured domains, such as human faces. The question, looking forward, is how to capture more data variability while keeping a high-degree of regularity in the input data representation? We believe that using programs as a data representation is the best avenue of attack, here. As we have shown in our work, a single program can capture a wide range of parametrically related shapes. One program, many shapes; strong regularity, but also high variability. We are excited to investigate extensions of ShapeAssembly, as well as other shape-generating languages, which can capture even more shape variability with highly-regular structures.

While ShapeAssembly has a strong inductive bias for generating physically-connected shapes, it is not guaranteed to do so. Hierarchical part structures which are locally connected everywhere may occasionally still exhibit disconnected leaf cuboids. This is more likely to happen with very non-axis-aligned structures that result in

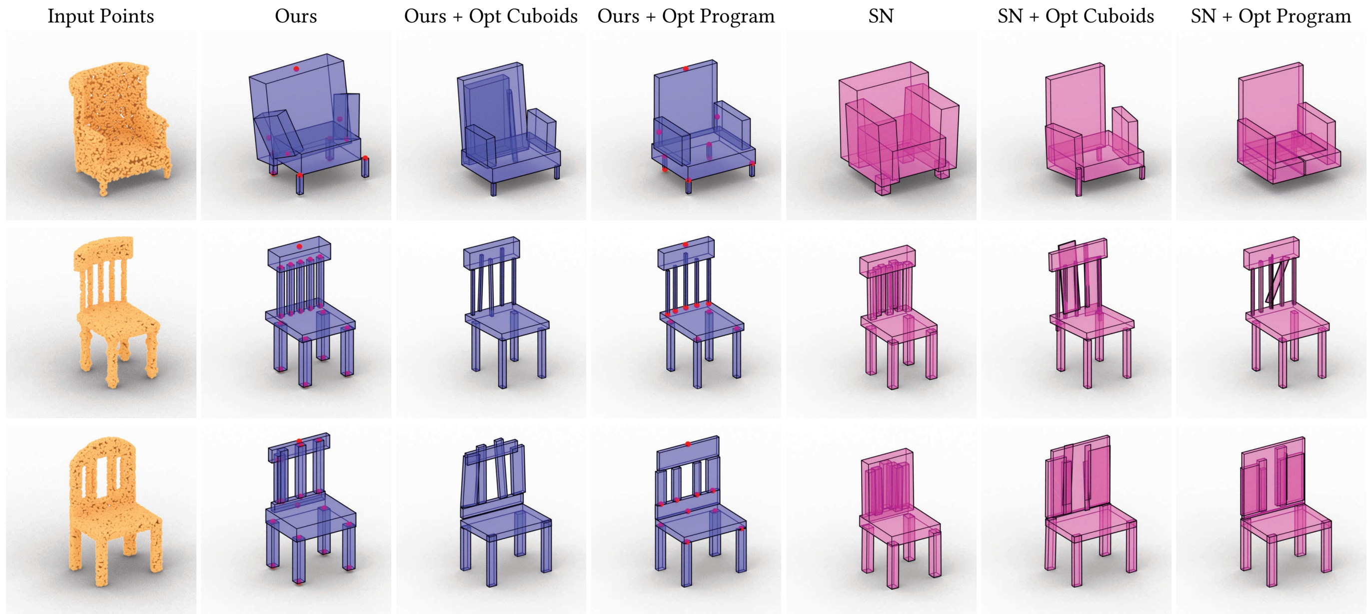

Fig. 13. Qualitative comparison of synthesis from point clouds of our method against StructureNet (SN). Our method is able to infer good program structures that match well with the unstructured geometry. The continuous parameters of this program structure can be further refined through an optimization procedure in order to better fit the target point cloud without creating artifacts.

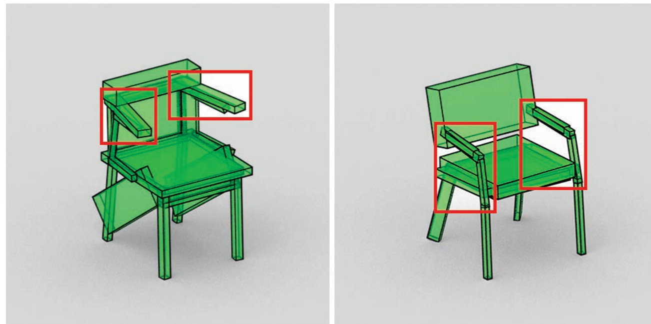

Fig. 14. Examples of PartNet shapes that contain parts whose orientations cannot be inferred from part-to-part attachments alone. While these shapes can be represented with ShapeAssembly programs that attach parts to "floating" points within the bounding volume, such programs are not added to our training data during our program extraction phase. As a result, our generative model never learns to produce shapes that require this type of attachment pattern.

loose bounding cuboids at the intermediate levels of the hierarchy. It is worth investigating mechanisms to guarantee that ShapeAssembly programs maintain leaf-to-leaf connectivity.

Future work. In Section 7, we showed an example of refining our generated hierarchical cuboid structures with point cloud surface geometry. This is not a new idea; other recent related work on part-based shape generation takes a similar approach to refining high-level part structures [Gao et al. 2019; Li et al. 2017; Mo et al. 2019a]. However, there is more work to be done at the intersection of structure generation and surface generation. These two paradigms could be much more closely married than they have been thus far, as existing part-surface generation has been explored largely in an independent, part-by-part fashion. What would it look like to swap out the "surface style" code for a shape while retaining its "structure" code? Our procedural representation may confer distinct advantages here, as the attachments explicitly specify where and how part geometries must connect.

It would also be interesting to move beyond cuboids as the proxy geometry used for atomic parts, as not all atomic parts are well-approximated by rectilinear geometry. In some cases, spherical, cylindrical, or more general curvilinear geometry would be a better choice. Pursuing this direction would help push more shape variability into the procedural representation, so that we do not lean so heavily on the neural network to capture it.

Another way to push knowledge from the learned latent space into the programs would be to make the programs include constraints on their parameters: either independent bounds, or correlations between parameters. For instance, it is non-semantic to make a chair leg too thin, or to make a chair back much narrower than the seat to which it is attached. It should be possible to mine shape datasets for this information, and to include it in the data used to train the generative model.

There are also more opportunities to apply generative models of shape programs to "synthesis from X" applications. While we showed translation from point clouds to shape programs, there are many more exicting possibilities in terms of linking 3D geometry,

2D images, and shape programs, and seamlessly using the three modalities to author different forms of shape manipulations.

Finally, if we aim for our generated shapes to be useful in embodied AI applications, they should also be equipped with information about kinematics and/or dynamics. For example, a program which specifies a cabinet could also specify the type of hinge with which the door attaches to the body, and how far that hinge opens. Ultimately, we believe that shape programs, and generative models which produce them, are the right fundamental representation for both human creative tasks and AI analysis tasks involving part-based 3D shapes.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful suggestions. Renderings of part cuboids and point clouds were produced using the Blender Cycles renderer. This research was supported by the National Science Foundation (#1753684, #1941808), a Brown University Presidential Fellowship, gifts from the University of College London AI Center and Adobe Research, and by GPU donations from NVIDIA. Daniel Ritchie is an advisor to Geopipe, Inc. and owns equity in the company. Geopipe is a start-up that is developing 3D technology to build immersive virtual copies of the real world with applications in various fields, including games and architecture.

REFERENCES

Ben Abbatematteo, Stefanie Tellex, and George Konidaris. 2019. Learning to Generalize Kinematic Models to Novel Objects. In *Proceedings of the Third Conference on Robot Learning*.

Zhiqin Chen, Andrea Tagliasacchi, and Hao Zhang. 2019. BSP-Net: Generating Compact Meshes via Binary Space Partitioning. arXiv:cs.CV/1911.06971

Zhiqin Chen and Hao Zhang. 2019. Learning Implicit Fields for Generative Shape Modeling. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

İ. Demir, D. G. Aliaga, and B. Benes. 2016. Proceduralization for Editing 3D Architectural Models. In *2016 Fourth International Conference on 3D Vision (3DV)*.

Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. 2018. InverseCSG: Automatic Conversion of 3D Models to CSG Trees. *ACM Trans. Graph.* 37, 6 (Dec. 2018).

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. 2019. Write, Execute, Assess: Program Synthesis with a REPL. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Haoqiang Fan, Hao Su, and Leonidas J Guibas. 2017. A point set generation network for 3D object reconstruction from a single image. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 605–613.

Lin Gao, Jie Yang, Tong Wu, Yu-Jie Yuan, Hongbo Fu, Yu-Kun Lai, and Hao (Richard) Zhang. 2019. SDM-NET: Deep Generative Network for Structured Deformable Mesh. In *SIGGRAPH Asia*.

Thibault Groueix, Matthew Fisher, Vladimir G. Kim, Bryan C. Russell, and Mathieu Aubry. 2018. AtlasNet: A Papier-Mâché Approach to Learning 3D Surface Generation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. 2017. GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium. In *NeurIPS*.

Irvin Hwang, Andreas Stuhlmüller, and Noah D. Goodman. 2011. Inducing Probabilistic Programs by Bayesian Program Merging. *CoRR* arXiv:1110.5667 (2011).

Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Judy Hoffman, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. 2017. Inferring and Executing Programs for Visual Reasoning. In *ICCV*.

Amlan Kar, Aayush Prakash, Ming-Yu Liu, Eric Cameracci, Justin Yuan, Matt Rusiniak, David Acuna, Antonio Torralba, and Sanja Fidler. 2019. Meta-Sim: Learning to Generate Synthetic Datasets. arXiv:cs.CV/1904.11621

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014).

Diederik P. Kingma and Max Welling. 2014. Auto-Encoding Variational Bayes. In *International Conference on Learning Representations (ICLR)*.

Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. 2017. Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction. *ACM Transactions on Graphics* 36, 4 (2017).

Eric Kolve, Roozbeh Mottaghi, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. 2017. AI2-THOR: An Interactive 3D Environment for Visual AI. *CoRR* arXiv:1712.05474 (2017).

Matt J. Kusner, Brooks Paige, and José Miguel Hernández-Lobato. 2017. Grammar Variational Autoencoder. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70 (ICML'17)*. JMLR.org, 1945–1954.

Manfred Lau, Akira Ohgawara, Jun Mitani, and Takeo Igarashi. 2011. Converting 3D Furniture Models to Fabricatable Parts and Connectors. *ACM Trans. Graph.* 30, 4, Article 85 (July 2011), 6 pages. https://doi.org/10.1145/2010324.1964980

Jun Li, Kai Xu, Siddhartha Chaudhuri, Ersin Yumer, Hao Zhang, and Leonidas Guibas. 2017. GRASS: Generative recursive autoencoders for shape structures. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 52.

Yunchao Liu, Zheng Wu, Daniel Ritchie, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. 2019. Learning to Describe Scenes with Programs. In *International Conference on Learning Representations (ICLR)*.

Sidi Lu, Jiayuan Mao, Joshua B. Tenenbaum, and Jiajun Wu. 2019. Neurally-Guided Structure Inference. In *International Conference on Machine Learning (ICML)*.

Andrew L. Maas. 2013. Rectifier Nonlinearities Improve Neural Network Acoustic Models.

A. Martinovic and L. Van Gool. 2013. Bayesian Grammar Learning for Inverse Procedural Modeling. In *CVPR*.

Mateusz Michalkiewicz, Jhony K. Pontes, Dominic Jack, Mahsa Baktashmotlagh, and Anders P. Eriksson. 2019. Deep Level Sets: Implicit Surface Representations for 3D Shape Inference. *CoRR* abs/1901.06802 (2019).

Kaichun Mo, Paul Guerrero, Li Yi, Hao Su, Peter Wonka, Niloy Mitra, and Leonidas Guibas. 2019a. StructureNet: Hierarchical Graph Networks for 3D Shape Generation. In *SIGGRAPH Asia*.

Kaichun Mo, Shilin Zhu, Angel X. Chang, Li Yi, Subarna Tripathi, Leonidas J. Guibas, and Hao Su. 2019b. PartNet: A Large-Scale Benchmark for Fine-Grained and Hierarchical Part-Level 3D Object Understanding. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. 2006. Procedural Modeling of Buildings. In *SIGGRAPH*.

Gen Nishida, Adrien Bousseau, and Daniel G. Aliaga. 2018. Procedural Modeling of a Building from a Single Image. *Computer Graphics Forum (Eurographics)* 37, 2 (2018).

Gen Nishida, Ignacio Garcia-Dorado, Daniel G Aliaga, Bedrich Benes, and Adrien Bousseau. 2016. Interactive Sketching of Urban Procedural Models. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 130.

Yoav I. H. Parish and Pascal Müller. 2001. Procedural Modeling of Cities. In *SIGGRAPH*.

Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. 2019. DeepSDF: Learning Continuous Signed Distance Functions for Shape Representation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

Przemyslaw Prusinkiewicz and Aristid Lindenmayer. 1996. *The Algorithmic Beauty of Plants*. Springer-Verlag, Berlin, Heidelberg.

Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. 2017. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *Advances in neural information processing systems*. 5099–5108.

Stephan R Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. 2016. Playing for data: Ground truth from computer games. In *European conference on computer vision*. Springer, 102–118.

Daniel Ritchie, Sarah Jobalia, and Anna Thomas. 2018. Example-based Authoring of Procedural Modeling Programs with Structural and Continuous Variability. In *EUROGRAPHICS*.

Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, Devi Parikh, and Dhruv Batra. 2019. Habitat: A Platform for Embodied AI Research. In *The IEEE International Conference on Computer Vision (ICCV)*.

Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. 2018. CSGNet: Neural Shape Parser for Constructive Solid Geometry. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Ondrej Stava, Bedrich Benes, Radomír Mech, Daniel G. Aliaga, and Peter Kristof. 2010. Inverse Procedural Modeling by Automatic Generation of L-systems. *Comput. Graph. Forum* 29 (2010), 665–674.

Minhyuk Sung, Hao Su, Vladimir G. Kim, Siddhartha Chaudhuri, and Leonidas Guibas. 2017. ComplementMe: Weakly-Supervised Component Suggestions for 3D Modeling. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* (2017).

Jerry O. Talton, Lingfeng Yang, Ranjitha Kumar, Maxine Lim, Noah D. Goodman, and Radomír Mech. 2012. Learning design patterns with Bayesian grammar induction. In *UIST*.

Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. 2019. Learning to Infer and Execute 3D Shape Programs. In *International Conference on Learning Representations (ICLR)*.

Jiajun Wu, Chengkai Zhang, Tianfan Xue, William T. Freeman, and Joshua B. Tenenbaum. 2016. Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Fei Xia, Amir R. Zamir, Zhi-Yang He, Alexander Sax, Jitendra Malik, and Silvio Savarese. 2018. Gibson env: real-world perception for embodied agents. In *CVPR*.

A. Khosla F. Yu L. Zhang X. Tang J. Xiao Z. Wu, S. Song. 2015. 3D ShapeNets: A Deep Representation for Volumetric Shapes. In *Computer Vision and Pattern Recognition*.

Yinda Zhang, Shuran Song, Ersin Yumer, Manolis Savva, Joon-Young Lee, Hailin Jin, and Thomas Funkhouser. 2017. Physically-Based Rendering for Indoor Scene Understanding Using Convolutional Neural Networks. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017).

Chenghui Zhou, Chun-liang Li, and Barnabas Poczos. 2019. Program Synthesis for Images using Tree-Structured LSTM. In *PGR Workshop at NeurIPS*.

Chenyang Zhu, Kai Xu, Siddhartha Chaudhuri, Renjiao Yi, and Hao Zhang. 2018. SCORES: Shape Composition with Recursive Substructure Priors. *ACM Transactions on Graphics (TOG)* 37, 6 (2018), 211:1–211:14.

Chuhang Zou, Ersin Yumer, Jimei Yang, Duygu Ceylan, and Derek Hoiem. 2017. 3D-PRNN: Generating Shape Primitives with Recurrent Neural Networks. In *IEEE International Conference on Computer Vision (ICCV)*.

A SEMANTICS OF THE ATTACH COMMAND

In designing the ShapeAssembly interpreter, our goal is to ensure that its internal operations stay limited to simple fixed-function, differentiable operations. Thus, implementing the `attach` command, we opt not to use any constrained optimization routines which could resolve a globally-optimal configuration of cuboids given the attachment constraints. Instead, the interpreter immediately executes each attachment as it is declared, i.e. it greedily solves for attachments. To make the behavior of this procedure as predictable as possible, the greedy attachment procedure should induce the fewest changes possible to the current cuboid shapes.

With these desiderata in mind, we designed the following procedure for attaching cuboid c_1 to cuboid c_2 (see Figure 15). The logic that executes depends upon how many prior attachments c_1 has and the aligned flag of c_1:

No prior attachments. In this case, cuboid c_1 can connect to cuboid c_2 by simply translating until the attach points are colocated.

One prior attachment. Here, the interpreter scales cuboid c_1 along one of its axes and then rotates it such that the attachment is satisfied. To choose the axis along which to scale c_1, the interpreter checks how quickly scaling each of its three dimensions would reduce the ratio n/k, where n is the distance between c_1's existing attachment point and the new target attachment point, and k is the distance between c_1's existing attachment point and the new source attachment point. The interpreter then scales c_1 by n/k along this dimension, which gives it the correct length. Finally, c_1 is rotated such that the source and target attachment points are colinear (and thus colocated).

Two or more prior attachments. In this case, it is not always possible to satisfy the attachment, as three point constraints on a cube may be overconstrained. If a solution exists, however, our interpreter will find it. And in the case where no solution exists, it attempts to approximately satisfy the attachment (which we decided to be more user-friendly behavior than throwing an error).

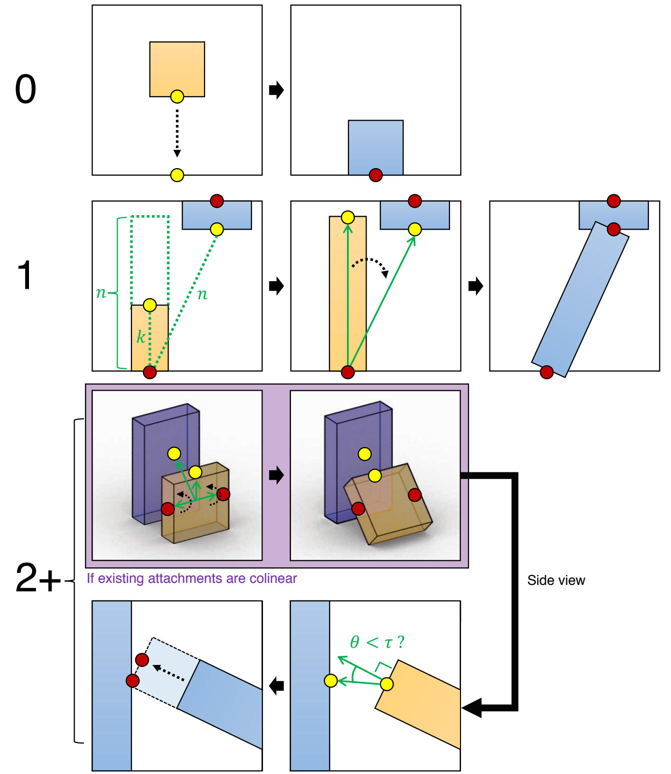

Fig. 15. Illustrating how the `attach` command executes, depending on the number of existing attachments (left column) to the cuboid in question. Cuboids with no existing attachments can simply be translated into place (top). Cuboids with one existing attachment can be scaled along one axis and then rotated (middle). Cuboids with two or more existing attachments are more complicated, and the attachment may not always be satisfiable. Our interpreter attempts to rotate and scale the cuboid to get as close as possible to valid solution.

First, the interpreter checks if c_1's existing attachment points are all colinear. If they are, then it rotates c_1 about this axis of colinearity to make the source attachment point face the target attachment point. The final step is to scale c_1 along the normal of the face containing the source attachment point. If the existing attachment points were not colinear, and this face was not rotated to point toward the target attachment point, then this may not be a useful operation (i.e. it may introduce undesirable change to the cuboid shape while doing little to bring the source point closer to the target point). Thus, the interpreter only executes this scale if the angle between the source face normal and the vector to the target point is smaller than a threshold τ (25 degrees in our implementation).

Aligned Cuboids. Cuboids that are marked as aligned in ShapeAssembly programs cannot have their orientations changed through attachment. In fact, with correct cuboid dimension parameterization, a single attachment is enough to properly position and orient an aligned cuboid. However, in order to ensure that aligned Cuboids remain connected through edits and predictions of our generative model, we minimally grow aligned cuboid dimensions to satisfy the

part-to-part connectivity specified through attachments. That is, for aligned cuboids we do not guarantee attachment point colocation after the first attachment, as this is often impossible to exactly fulfill without changing a cuboid's orientation. Rather, we guarantee that aligned cuboids will fulfill attachment relationships with cuboids they are attached to at *some* attachment point.

B SEMANTICS OF SHAPEASSEMBLY MACRO FUNCTIONS

We provide an account of the logic for macro function expansion in ShapeAssembly :

Squeeze. The `squeeze` macro is parameterized by three cuboids (c_{n1}, c_{n2}, c_{n3}) a face f and a (u, v) position on f's 2D coordinate system. A `squeeze` command expands into two `attach` functions. The first attach function attaches the center of c_{n1}'s f face to the (u, v) position on the opposite face of f on c_{n2}. The second attach function attaches the center of c_{n1}'s opposite face of f to the (u, v) position on the face of f on c_{n3}. For example, the line `squeeze` (c_{n1}, c_{n2}, c_{n3}, left, .1, .4). It expands into `attach`(c_{n1}, c_{n2}, 0.0, .5, .5, 1.0, .1, .4) and `attach`(c_{n1}, c_{n3}, 1.0, .5, .5, 0.0, .1, .4).

Reflect. The `reflect` macro is parameterized by a cuboid c_n and an axis a. A `reflect` command first expands into one `Cuboid` function, that creates a new cuboid $c_{n'}$ with the same parameters as c_n. Then for every previous attachment line pair that had moved c_n, of the form `attach`$(c_n, c_m, x_1, y_1, z_1, x_2, y_2, z_2)$, the reflect command creates a new attachment line: `attach`$(c_{n'}, c_m, x_1, y_1, z_1, R(x_1, y_1, z_1, c_n, c_m, a))$. R is a function that applies a reflection of the global point specified by (x_1, y_1, z_1) in the local coordinate frame of c_n about the axis a, and then returns the local coordinates of that point within c_m.

Translate. The `translate` macro is parameterized by a cuboid c_n, an axis a, a number of members m, and a distance d. A `translate` command first expands into m `Cuboid` functions, that each creates a new cuboid c_{n_i} with the same parameters as c_n. Then for every previous attachment line pair that had moved c_n, of the form `attach`$(c_n, c_m, x_1, y_1, z_1, x_2, y_2, z_2)$, the translate command creates a new attachment line `attach`$(c_{n_i}, c_m, x_1, y_1, z_1, T(x_1, y_1, z_1, c_n, c_m, a, d))$. T is a function that applies a translation of the global point specified by (x_1, y_1, z_1) in the local coordinate frame of c_n along the axis a (of the bounding volume) for for a distance of d (where d is normalized by the size of the bounding volume), and then returns the local coordinates of that point within c_m.

C PROGRAM EXTRACTION PROCEDURE

Here, we provide an account of our program extraction procedure in greater detail:

Part Shortening. Before any hierarchical processing, we first attempt to regularize any artifacts in the input data. Specifically, for each leaf cuboid part proxy, we check if any of its faces are completely contained within any other leaf cuboid. If we find that we can shorten a leaf cuboid without changing the visible, non-intersecting, geometry of the part graph, we do so.

Semantic Hierarchy Arrangement. During our data preprocessing stage when converting PartNet part graphs into ShapeAssembly programs, we locally flatten part graph hierarchies based on semantic rules as depicted in Figure 5. For chairs we flatten the following nodes: back, arm, base, seat, footrest and head. For tables we flatten the following nodes: top and base. For storage we flatten the following nodes: cabinet frame, cabinet base. For storage, we move the following nodes into the cabinet frame sub-program: countertop, shelf, drawer, cabinet door and mirror. We also perform a semantic collapsing step where the intermediate nodes containing detailed geometry are converted into leaf nodes and their children are discarded. For chairs we collapse the following nodes: caster and mechanical control. For tables we collapse the following nodes: caster, cabinet door, drawer, keyboard tray. For storage we collapse the following nodes: drawer, cabinet door, mirror and caster. Empirically we observed that this method of hierarchy re-arrangements produces cleaner and more regularized training data for our generative model.

Attachment Point Detection. In order to identify which cuboids connect, and where they connect, we use a point cloud intersection procedure. We sample a uniform 20x20x20 point cloud within the volume defined by each cuboid. To check if two cuboids are attached, we find the set of points in the pairwise point cloud comparison that have a minimum distance to any point in the other point cloud within a distance threshold determined by the scale of the larger cuboid. For cuboids that attach (i.e. this intersection set is non-zero) we sample a denser 50x50x50 point cloud within the bounds of the detected intersection volume, forming a set of candidate attachment points. From this set we first filter all attachment points that are outside of either cuboid. If any remaining attachment points form face-to-face connections between cuboids we choose them, otherwise we define the attachment as taking place at the mean of the remaining attachment points. With the same procedure, we also record if cuboids connect to the top or bottom of the bounding volume. Sampled points with bounding volume local y-coordinates in the ranges of $[0, 0.05]$ and $[.95, 1.0]$ are assigned to the bottom and top respectively.

Symmetry Detection. We enforce that all members of a symmetry group share the same connectivity structure in the input part graph. Cuboids are grouped together by symmetry if they: (i) connect to the same cuboids, (ii) share a reflectional or translational symmetry about the X, Y or Z axis of their parent bounding volume, and (iii) each attachment point involved in their outgoing connections also shares this same symmetrical relationship. Two cuboids, or two attachment points, are considered to share a symmetrical relationship if applying the symmetry transformation matrix to one member produces a parameterization close to that of the other member.

Notice that this procedure can disqualify symmetry formation about groups of interconnected cuboids that share a symmetrical relationship. As such, before forming symmetry groups about individual cuboids, we attempt to form symmetry groups about connected components of multiple cuboids. Whenever such a component is found, we locally abstract its structure with a bounding volume, and create a symmetry group sub-program. In this manner, we capture

additional spatial symmetries while continuing to enforce the relationship between symmetry and part connectivity. The "H-leg" program (Program3) in Figure 2 shows an example of where such a symmetry sub-program was formed.

In total, our parsing procedure finds valid ShapeAssembly programs for 46% of Chairs, 65% of Tables and 58% of Storage shapes in PartNet.

D DECODER SEMANTIC VALIDITY CHECKS

During the process of decoding a latent code, our generative network enforces the following semantic validity conditions on its outputs:

- XYZ attachment coordinates are clamped between 0 and 1.0. Additionally, attachments to the bounding box can only be at the top or bottom faces with an allowable error of .05.
- Cuboid dimensions are clamped between 0.01 and the corresponding bounding box dimension
- Bounding box cuboids can have no sub-programs
- Cuboids only attach at a single location. As an exception, cuboids are allowed to attach to both the top and bottom faces of the bounding volume.
- The bounding box cannot be moved by an attach command
- Attachment orderings must be grounded. Upon terminating, any ungrounded cuboids instantiations are discarded.
- Symmetries can only operate on grounded cuboids
- The ordering of Cuboid, attach, squeeze, reflect, and translate lines must be consistent with the ShapeAssembly grammar.
- Commands must keep cuboids within the bounds of the defined bounding volume with an allowable error of 10%.

During generation, if our model predicts a non-semantic program line, we attempt to back-track until we are able to find a semantically valid solution. For instance, if we predict a new line to be a `reflect` command, but no cuboids have been grounded, we pick a new command type for the line by zeroing out the logits for the reflect command index.

In some cases, a combination of bad continuous parameters and program structure predictions produce a violating line that cannot be easily fixed. During unconditional generation, we reject the sample if we encounter this behavior (this happens for 10% - 20% of our random samples across the categories we consider). We run an ablation on this rejection sampling in Table 2. During interpolation, we never reject a sample. Instead, we simply do not add lines to the predicted program for which we could not find a fix.

E SHAPE QUALITY METRICS

We provide additional details about the metrics used in Table 2:

- **Rootedness :** We check if a connected path exists between the ground and all parts in the shape. We judge two parts to be connected if they are separated by a distance no larger than 2% of the overall shape's bounding box diagonal length.
- **Stability :** We convert generated 3D shape structures into rigid bodies and place them in a physical simulation with gravity. A vertical force is applied to each shape proportional to its mass, along with some other small random forces and torques. If the resting height of any connected component of the shape changes by more than 10% after these perturbations we declare it unstable. Note that this is by definition less than or equal to the percentage of rooted shapes, as a shape must be rooted in order to be stable.
- **Realism:** The percentage of test set shapes classified as "generated" by a binary PointNet classifier trained to distinguish between generated shapes and shapes from the training dataset. The classifier is trained on an equal amount of positive and negative examples for 300 epochs. We hold out a portion of shapes from the test set, and measure the percentage of them incorrectly classified as "fake". To reduce fluctuation, the percentage is averaged over the last 50 epochs.