Supplemental Materials for The Shape Part Slot Machine: Contact-based Reasoning for Generating 3D Shapes from Parts

Kai Wang[1], Paul Guerrero[2], Vladimir Kim[2], Siddhartha Chaudhuri[2], Minhyuk Sung[2,3], and Daniel Ritchie[1]

[1] Brown University
[2] Adobe Research
[3] KAIST

1 Data Preparation

We use the PartNet [5] dataset for all our experiments, following the train/validation/test split provided in the original paper.

1.1 Obtaining Part Level Geometry

Each shape in the PartNet data comes with a semantic hierarchy that decomposes the shape into parts in a coarse-to-fine manner. We use the finest-grained level of parts in this hierarchy. We filter the data using the following criteria:

- We remove shapes that contain only 1 part or more than 30 parts.
- We detect inconsistent shapes with parts that do not equal the union of their children. For the chairs and tables dataset, we reject these inconsistent shapes. The furniture and lamps dataset are smaller, so in these datasets, we keep the inconsistent parts, but discard their children.
- We remove shapes with parts that contain floating geometry due to annotation errors. We detect such cases by first clustering the part's point cloud with DBSCAN [1]. If there exist any cluster that is significantly smaller than other clusters, we reject the entire shape. (We cannot reject all parts that consist of multiple disconnected clusters, because some parts contain multiple symmetric disconnected components).
- We remove shapes that are disconnected based on the adjacency edges we detect.

Table 1 summarizes the size of the dataset before and after filtering.

1.2 Extracting Relationships Between Parts

After obtaining the geometry of individual parts, we sample the surface of each part uniformly to obtain a 3000-point representation. We then detect relationships between parts based on the protocol of StructureNet [4]:

Detecting Symmetry: We detect symmetry based on the methods proposed by Wang et al. [7]. We restrict the symmetry types to translational symmetry, reflectional symmetry about planes parallel to the three coordinate planes, and 4-way rotational symmetry

Table 1: Dataset statistics before and after our filtering process

Category	Split	Before	After
Chair	Train	4489	3315
	Val	617	438
	Test	1217	886
Table	Train	5707	4254
	Val	843	637
	Test	1668	1257
Storage	Train	1588	1123
	Val	230	152
	Test	451	290
Lamp	Train	1554	1187
	Val	234	181
	Test	419	321

about the y(up)-axis. We create an undirected graph for each of the symmetry type, where every edge is a detected symmetry between a pair of parts. We treat each connected component in these graphs as a symmetry group.

Pruning Symmetry: We then prune the detected symmetries by enforcing that each part belongs to at most one symmetry groups. We prioritize larger groups. If two groups are of the same size, then we favor the simpler explanation: translational $>$ rotational $>$ reflectional.

Detecting Adjacency: We regard two parts, A and B, as adjacent if the smallest distance between their respective points clouds is less than $\tau = \theta \mathbf{r}$, where $\mathbf{r}$ is the average bounding sphere radius of the two parts. We first detect symmetries using $\theta = 0.05$ i.e. the orignal setting of StructureNet. We then do a second pass of adjacency detection for parts involved in symmetry groups in order to recover any undetected adjancies to a common neighbor: We set $\theta = 0.1$ if A belongs to a symmetry group (before pruning) and B is adjacent (using $\theta = 0.05$) to any other parts in the same symmetry group, vice versa; we further increase θ to 0.3 if the involved symmetry group has more than 3 parts and at least 3 other parts are adjacent to B, vice versa. We use the same threshold τ for computing the points for each slot (Section 4.2).

Pruning Adjacency: We then attempt to identify the set of adjancency relationships that best describe the part structure. Note that this might not be necessary for a dataset where the connections between parts are more clearly defined. We prune the adjacency edges using the following set of heuristics, applied in order. All heuristics are only applied if removing the edge does not disconnect the adjacency graph:

We first remove any edges between parts in the same symmetry group, prior to symmetry pruning.

We then identify all triplet of parts A, B, C that overlap at the same area, and thus pairwise adjacent. For each triplet, we check if there's an edge that we can prune, using the following heuristics, without loss of generality, applied in order:

- If B and C shares a common parent in the PartNet hierarchy and A has a different parent, then we store either AB or AC for deletion if we can break ties between them: we store the edge for the part that is either significantly farther from A, smaller in surface area, or with less adjacent parts. We do not store any edges if the ties between B and C cannot be broken.
- We store AC for deletion if the y(up)-coordinate of the centroid of B is between those of A and C, and is of at least a distance of 0.05 away from each.
- We store BC for deletion if the surface area of both part B and C is signifcantly smaller than that of part A, or if B and C has roughly the same area but A does not.

We then sort all the candidate edges for deletion, prioritizing on those detected with heuristics mentioned earlier, and then those belonging to parts with smaller surface areas.

After finding all the candidates edges, we iterate over them and delete edges, while respecting the detected symmetries. For each candidate AB, we check if A and/or B belongs to any symmetry groups. If A is in a symmetry group, then we include all other adjacency edges from any parts in that symmetry group to B. We do the same for B. We proceed to remove all these edges if the following conditions are met:

- Removing these edges does not disconnect the graph.
- Removing these edges does not disconnect a symmetry group from its most frequent neighbor i.e. the part that has the highest number of adjacency edges to parts in the symmetry group. If multiple such neighbors exist, we prefer to keep the edges to the neighbor that is not in any symmetry groups. If there are still multiple such neighbors, we keep only 1 of them, and allow deleting edges to the rest. A special case occurs when every neighbor to a symmetry group is adjacent to exactly one part in the group. This often occurs when a group of symmetrical parts are decomposed further into subparts (e.g. four symmetrical legs are decomposed into four legs and for leg wheels). In such cases, we regard every part in the adjacent symmetry group as a most frequent neighbor.
- Either A and B are still both connected to C in the original triplet, or if there exists other parts in the region where A, B and C overlaps and a path can be found from A to B via those parts or vice versa.

2 More Details on the "What" Module

We provide additional details on the **What to Attach?** module (Section 5) here.

Given the graph features $h_{\mathcal{G}'}, h_{\mathcal{G}'_{\text{target}}}$, the mixture density network (MDN) represents the conditional probability distribution $P(X|\mathcal{G}', V_{\text{target}}, X \in \mathbb{R}_{\text{emb}})$ as a mixture of N gaussians, with mixing coefficients $\pi_1 \dots \pi_N$, means $\mu_1 \dots \mu_N$ and standard deviations $\sigma_1 \dots \sigma_N$ respectively. The probability of any embedding X_C, then, can be expressed as

$$p(X_C) = \sum_{k=1}^{N} \pi_k \cdot \mathcal{N}(X_C \mid \mu_k, \sigma_k^2)$$

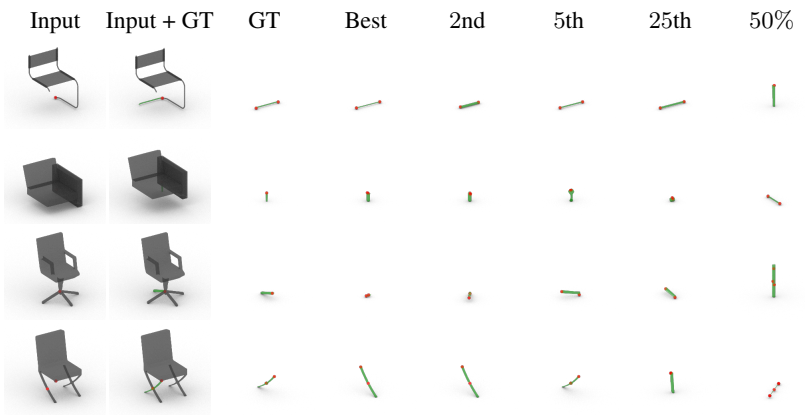

Fig. 1: Additional outputs of the **What to Attach?** module. We visualize the input partial slot graph within the parts that contain them (grey) and the center of the selected slots (red), as well as the ground truth part (green, 2nd column). The parts and slots are in their ground truth world-space pose, which is *not* available to the neural network. We then visualize, individually, the ground truth part and the retrieved candidates ranked 1st, 2nd, 5th, 25th, and at the 50th percentile, respectively, along with all of their slots (red).

We omit the conditions $(\mathcal{G}', V_{\text{target}})$ for simplicity of notation. In practice, we use negative log likelihood to setup the triplet loss:

$$\ell(X_C) = -\log \sum_{k=1}^{N} \pi_k \cdot \mathcal{N}(X_C \mid \mu_k, \sigma_k^2)$$

Given a positive example C_{target} and a negative example C_{negative}, we then obtain the final triplet loss as

$$\mathcal{L}(X_{C_{\text{target}}}, X_{C_{\text{negative}}}) = \max\{m + \ell(X_{C_{\text{target}}}) - \ell(X_{C_{\text{negative}}}), 0\}$$

Where m is a constant margin. We select the negative examples C_{negative} at training time by computing the triplet loss between the positive example and a set of randomly sampled negative examples, and choose one that gives a non-zero loss, whenever possible (i.e. using only semi-hard triplets).

In Figure 1, we provide additional examples of the learned module on chairs (see also Figure 4 in the main paper). The first row shows another example of query that demands a very specific type of structure. The second row shows another example of a query that asks for chair legs. We show failure cases of our module in the last 2 rows, where it fails to reason about the exact spatial structure of shapes and retrieves parts that are oriented incorrectly.

3 Generating New Slot Graphs at Test Time

Although trained on all shapes with less than or equal to 30 parts, less than 5 percent of the training shapes have more than 20 parts, and each of those shapes have rather unique structures. Therefore, when generating new slot graphs, we only use parts from

shapes with less than 20 parts. We start each shape by randomly selecting a part from the candidate parts. We then iteratively query the three neural network modules, until the slot graph is complete (when all slots are attached). During this generation process, we use the output of the three neural modules to detect and reject partial slot graphs that are outliers:

- If the **Where** module gives a probability p_{continue} of less than 0.5 when there are no slots selected.
- Not all part cliques retrieved by the What module are good candidates. We reject a retrieved candidate C_{target} if $|V_{\text{target}}| > |C_{\text{target}}|$, or if one of the edge predicted by the **Where** module has a probability less than $1/(\max(|V_{\text{target}}|+1, |C_{\text{target}}|)+0.5)$. If all candidates within a margin of 60 (100 for parts invovled in symmetry) from the highest scoring candidate are rejected, we reject the partial slot graph.

We also reject the generated slot graph if any of the following conditions are met:

- The slot graph contains more than 20 parts.
- The slot graph is detected as an outlier. We perform outlier detection using one-class Support Vector Machines (OCSVM). We fit one OCSVM for all graphs in the training set with the same number of parts. For each OCSVM that fits graphs with N nodes, we use a feature size of $3(N-1)$, with the following features:
  - Number of parts with an (adjacency) degree of $1 \ldots N-1$.
  - Number of parts where $1 \ldots N-1$ other parts are within a distance of 2.
  - Number of parts where the furthest part has a distance of $1 \ldots N-1$.

  Other commonly used graph summary statistics, such as clustering coefficient, number of n-cycles, etc. are also possible candidates here, but we found the set of features we use to be sufficient for our purposes.

4 Implementation Details

We set the rounds of message passing, T, to 10 for all our graph neural networks (GNN) operating on partial slot graphs. We set $T = 4$ for GNNs operating on part cliques. Since no adjacency edges exists, this effectively leads to 2 rounds of message passing. We set the dimension of node embeddings to 64 and the dimension of graph embeddings to 128. All MLP we use have 2 hidden layers and uses leaky ReLU as the activation function. We use a mixture of 10 gaussians for the MDN and a margin $m = 20$ for the triplet loss. We train all neural networks with the Adam [3] optimizer, and with a batch size of 32. We select the negative examples for the **What** module from 32 randomly selected slot graphs as well, for each training step.

5 Details on Baselines

We provide additional details on how we implemented the baselines.

ComplementMe: We re-implemented ComplentMe [6] in PyTorch. We mostly used the original settings of ComplementMe, with the following exceptions:

- We set the maximum threshold for the standard deviation of the Gaussian Mixture model to 50 instead of 0.05, since we found that the standard deviation of all Gaussians saturate at the original threshold very quickly.

- ComplementMe sampled random triplets originally, we instead sample only the semi-hard triplets i.e. triplets that give a non-zero training loss.
- In the paper, ComplementMe suggests that the placement networks do not share weights with the retrieval/embedding networks. This is not the case in their official implementation. We followed the description in the paper.
- We removed all BatchNorm layers from the PointNet backbone since we observed that including them hurts the evaluation performance.

We train ComplementMe until convergence.

StructureNet: We use the pre-trained models provided by StructureNet [4], which are trained on the same split as what we use in the paper. Do note that StructureNet uses a different data filtering strategy than ours, so the training set will differ slightly. We encode every part in the test set using the pre-trained part encoder, with each part centered and normalized in the same way as they would be if used to train StructureNet. We then use the provided evaluation script to randomly sample outputs. Instead of decoding child-level latent code to point clouds, we directly retrieve the test set part that is the closest in the latent space, and then apply the predicted transformations to the retrived part.

6 More Results

We show random samples of our method and the baselines on all four categories in Figure 2, 3, 4 and 5.

For the methods that are autoregressive (ours and ComplementMe), the color of the parts correlate with the order in which they are inserted. We use the Tableu 10 color palette[4] for the first 10 parts, and add the remaining 10 colors in the Tableu 20 color palette for shapes more than 10 parts: the blue part is used for initialization, and the subsequent parts inserted are colored orange, green, red, purple, etc. respectively.

Overall, the quality and physical plausibility of the generated shapes correlate well with the quantitative metrics. ComplementMe benefits considerably from grouping parts by symmetry, as it simplifies the task of predicting global poses of shapes significantly. When parts are not grouped by symmetry, it often fails to predict the right pose of parts, and sometimes is not able to complete a shape at all. It also produces a lot of incorrect and incomplete storages, even with the help of symmetry.

StructureNet is usually able to generate shapes that are plausible, though often with a few missing parts. However, it has the tendency to generate only a subset of shape types. This is most apparent for table and storage, when it generates mostly square tables and storages with open shelves. Large gaps sometimes exist between the individual parts, leading to problems with physical plausibility. Note that this problem is not caused by us retrieving parts directly using the latent code — the box version of StructureNet has similar issues (see the evaluation of ShapeAssembly [2]).

The behavior of our method is more polarizing: it generates a lot of high quality shapes; however, some other generated shapes are totally incorrect. The high quality

[4] https://public.tableau.com/views/TableauColors/ColorPaletteswithRGBValues

shapes fare better than the baselines in terms of quality, physical plausibility, and diversity to some extent. The incorrect shapes exhibit a wide range of failure mode, which we hypothesize can be traced back to a few incorrect steps in the autoregressive generation process. Reducing the chance of these incorrect steps, and identifying them when they happen, is an important future direction to take in order to further improve the quality of the generated shapes. Our method also has the tendency to generate simpler shapes when sampling randomly. This is not caused by the neural networks learning biased distributions, but caused by the higher failure rate for more complex structure during autoregressive sampling. We also notice a few repeated shapes, especially for storages. This can be addressed by sampling the neural network modules randomly, as opposed to doing MAP inference. In figure 6, we show examples of random sampling: our method is able to produce multiple output per initialization (blue).

Finally, we show random samples from drawn the test set in Figure 7. We note that none of the methods are able to generate shapes that are close to the dataset in terms of quality and diversity. This is especially the case for shapes with unique and complex structures: they are harder to learn, and there is often not enough training data for them. Learning these structures correctly and efficiently remains an open problem.

Fig. 2: Chair Unconditional Samples

Fig. 3: Table Unconditional Samples

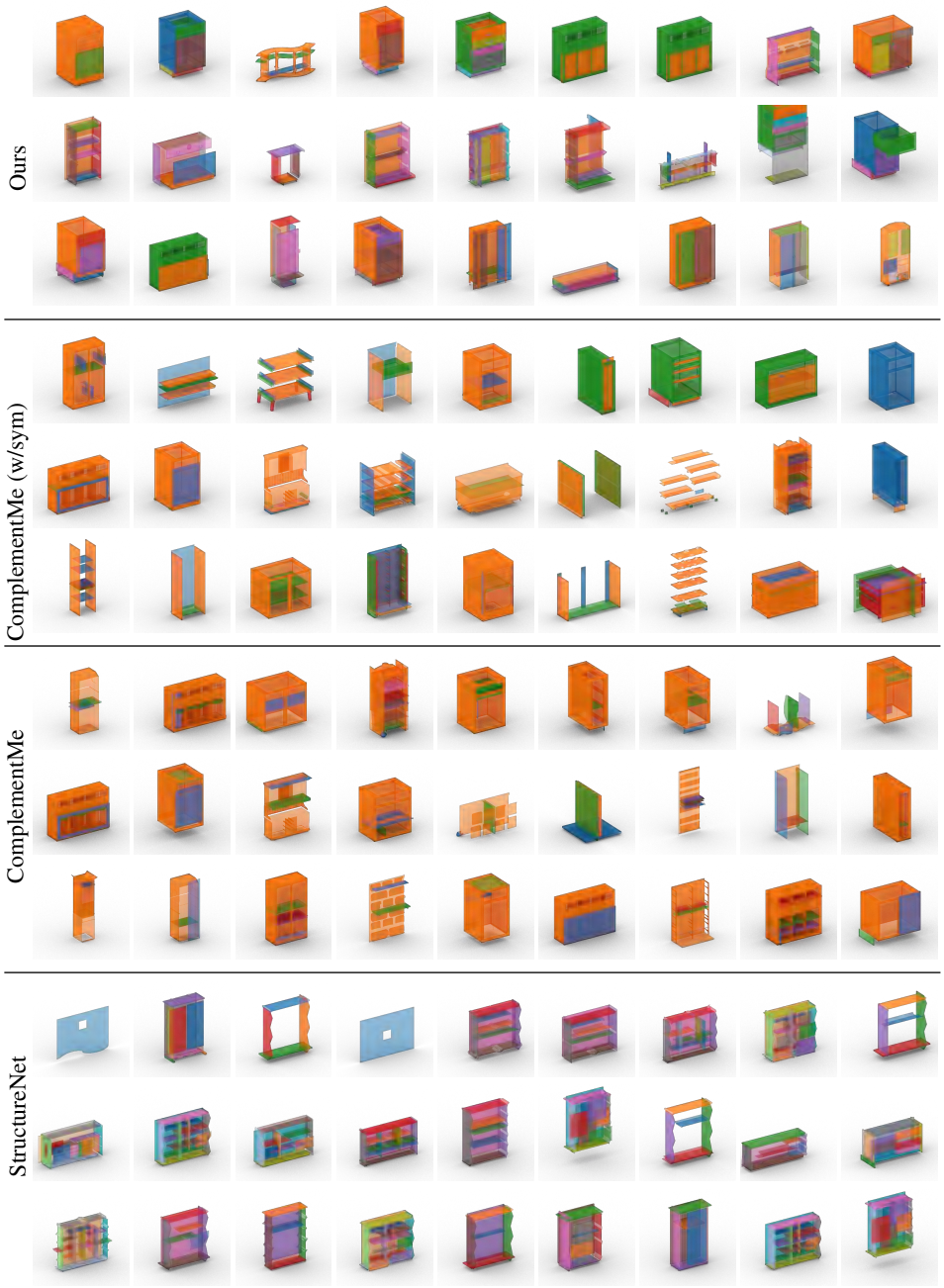

Fig. 4: Storage Unconditional Samples

Fig. 5: Lamp Unconditional Samples

Fig. 6: Multiple output per initialization, achieved by sampling the neural networks randomly instead of doing MAP inference. Each row uses a different part as initialization.

Fig. 7: Dataset Unconditional Samples

References

1. Ester, M., Kriegel, H.P., Sander, J., Xu, X., et al.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: kdd. vol. 96, pp. 226–231 (1996)
2. Jones, R.K., Barton, T., Xu, X., Wang, K., Jiang, E., Guerrero, P., Mitra, N.J., Ritchie, D.: Shapeassembly: Learning to generate programs for 3d shape structure synthesis. ACM Transactions on Graphics (TOG), SIGGRAPH Asia 2020 **39**(6), Article 234 (2020)
3. Kingma, D.P., Ba, J.: Adam: A Method for Stochastic Optimization. In: ICLR 2015 (2015)
4. Mo, K., Guerrero, P., Yi, L., Su, H., Wonka, P., Mitra, N., Guibas, L.: StructureNet: Hierarchical graph networks for 3D shape generation. In: SIGGRAPH Asia (2019)
5. Mo, K., Zhu, S., Chang, A.X., Yi, L., Tripathi, S., Guibas, L.J., Su, H.: Partnet: A large-scale benchmark for fine-grained and hierarchical part-level 3d object understanding. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 909–918 (2019)
6. Sung, M., Su, H., Kim, V.G., Chaudhuri, S., Guibas, L.: ComplementMe: Weakly-supervised component suggestions for 3D modeling. ACM Transactions on Graphics (TOG) **36**(6), 226 (2017)
7. Wang, Y., Xu, K., Li, J., Zhang, H., Shamir, A., Liu, L., Cheng, Z., Xiong, Y.: Symmetry hierarchy of man-made objects. In: Computer graphics forum. vol. 30, pp. 287–296. Wiley Online Library (2011)