Supplemental Material for PLAD: Learning to Infer Shape Programs with Pseudo-Labels and Approximate Distributions

A. Details of Domain Grammars

2D CSG We follow the grammar from CSGNet [3]. This grammar contains 3 Boolean operations (intersect, union, subtract), 3 primitive types (square, circle, triangle), and parameters to initialize each primitive (L and R tuples). Please refer to the CSGNet paper for details.

$$
\begin{aligned}
&S \rightarrow E;\\
&E \rightarrow EET \mid P(L,R);\\
&T \rightarrow \mathit{intersect} \mid \mathit{union} \mid \mathit{subtract};\\
&P \rightarrow \mathit{square} \mid \mathit{circle} \mid \mathit{triangle};\\
&L \rightarrow \left[8:8:56\right]^2;\;\; R \rightarrow \left[8:4:32\right].
\end{aligned}
$$

3D CSG We design our own grammar for 3D CSG similar in spirit to the grammar of CSGNet. While CSGNet does contain a 3D CSG grammar, we find that it overly discretizes the possible spacing and positioning of primitives. Therefore in our grammar, we allow each primitive to be parameterized at the same granularity as the voxel grid (32 bins). In this way, each primitive takes in 6 parameters (instead of 2 parameter tuples), where the 6 parameters control the position and scaling of the primitive.

$$
\begin{aligned}
&S \rightarrow E;\\
&E \rightarrow EET \mid P(F,F,F,F,F,F);\\
&T \rightarrow \mathit{intersect} \mid \mathit{union} \mid \mathit{subtract};\\
&P \rightarrow \mathit{cuboid} \mid \mathit{ellipsoid};\\
&F \rightarrow \left[1:32\right]
\end{aligned}
$$

ShapeAssembly ShapeAssembly is a domain-specific language for creating structures of 3D Shapes [1]. It creates structures by instantiating parts (*Cuboid* command), and then attaching parts to one another (*attach* command). It further includes macro operators that capture higher-order spatial patterns (*squeeze*, *reflect*, *translate* commands). To remain consistent with our CSG experiments, we further modify the grammar such that all continuous parameters are discretized.

$$
\begin{aligned}
&S \rightarrow BBoxBlock; ShapeBlock;\\
&BBoxBlock \rightarrow \text{bbox} = \texttt{Cuboid}(1.0, x, 1.0)\\
&ShapeBlock \rightarrow PBlock; ShapeBlock \mid None\\
&PBlock \rightarrow c_n = \texttt{Cuboid}(x,x,x); ABlock; SBlock\\
&ABlock \rightarrow \mathit{Attach} \mid \mathit{Attach}; \mathit{Attach} \mid \mathit{Squeeze}\\
&SBlock \rightarrow Reflect \mid Translate \mid None\\
&\mathit{Attach} \rightarrow \texttt{attach}(\mathit{cube}_n, f, uv, uv)\\
&\mathit{Squeeze} \rightarrow \texttt{squeeze}(\mathit{cube}_n, \mathit{cube}_n, face, uv)\\
&Reflect \rightarrow \texttt{reflect}(\text{axis})\\
&Translate \rightarrow \texttt{translate}(\text{axis}, m, x)\\
&f \rightarrow \mathit{right} \mid \mathit{left} \mid \mathit{top} \mid \mathit{bot} \mid \mathit{front} \mid \mathit{back}\\
&\text{axis} \rightarrow X \mid Y \mid Z\\
&x \in [1,32]/32.\\
&uv \in [1,10]^2/10.\\
&n \in [0,10]\\
&m \in [1,4]
\end{aligned}
$$

B. Details of Synthetic Pretraining

2DCSG We follow the synthetic pretraining steps from CSGNet and directly use their released pretrained model weights. Please refer to their paper and code for further details.

3DCSG We generate synthetic programs for 3D CSG with the following procedure. First, we sample K primitives, where K is randomly chosen between 2 and 12. To sample a primitive, we sample a center position within the voxel space, and then we sample a scale, such that the scale is constrained so that the primitive will not extend past the borders of the voxel grid. We then find if the bounding boxes of any two primitives overlap in space (using the position and scale of each primitive). We then construct a binary tree of

Boolean operations by randomly merging the K primitives together, until only one group remains. Each Boolean operation merges two primitive groups into a single primitive group. The type of semantically valid Boolean operation depends on the overlaps between primitives of the two groups. When a group of primitives A and a group of primitives B is merging: union is always a valid operation, difference is a valid operation if each primitive in group B shares an overlap with some primitive in group A, and intersection is a valid operation if each primitive in group A shares an overlap with some primitive in group B and each primitive in group B shares an overlap with some primitive in group A. We can then unroll this binary tree of boolean operations into a sequence of tokens from the CSG grammar, forming a synthetic program. We sample 2,000,000 synthetic programs according to this procedure, that are used during supervised pretraining, and we sample another 1000 synthetic programs that we use a validation set. We pretrain our model for 40 epochs, where each epoch takes around 1.5 hours to complete. At this check-point, the model had converged to a reconstruction IoU of 90 on both train and validation synthetic data.

ShapeAssembly We generate synthetic programs for ShapeAssembly with the following procedure. We first sample the number of primitive blocks K (PBlock), where K is randomly chosen between 2 and 8; note that the number of cuboids created can be greater then K, when symmetry operations are applied. Each PBlock is filled in with random samples according to the grammar syntax. First a cuboid is created, then an attach block is applied, then a symmetry block is applied. An attach block can contain either one attach operation, one squeeze operation, or two attach operations. A symmetry block can contain either a reflect operation, a translation operation, or no operation. Command parameters are randomly sampled according to simple heuristics (e.g. reflections are more common than translations) and in order to maintain language semantics (e.g. attaches can only be made to previously instantiated cuboid indices). A final validation step occurs after a complete set of program tokens has been synthetically generated; we execute the synthetic program, and check how many voxels are uniquely occupied by each cuboid in the executed output. If any cuboid uniquely occupies less than 8 voxels, the entire synthetic sample is rejected. We sample 2,000,000 synthetic programs according to this procedure, that are used during supervised pretraining, and we sample another 1000 synthetic programs that we use as a validation set. We pretrain our model for 26 epochs, where each epoch takes around 40 minutes to complete. At this check-point the model had converged to reconstruction IoU of 70 on both train and validation synthetic data.

C. Experiment Hyperparameters

3D Experiments For 3D CSG and ShapeAssembly, we use the following model hyper-parameters.

The encoder for both cases is a 3D CNN that consumes a 32 x 32 x 32 voxel grid. It has four layers of convolution, ReLU, max-pooling, and dropout. Each convolution layer uses kernel size of 4, stride of 1, padding of 2, with channels (32, 64, 128, 256). The output of the CNN is a (2x2x2x256) dimensional vector, which we transform into a (8 x 256) vector. This vector is then sent through a 3-layer MLP with ReLU and dropout to produce a final (8 x 256) vector that acts as an 8-token embedding of the voxel grid.

The decoder for both cases is a Transformer Decoder module [6]. It uses 8 layers and 16 heads, with a hidden dimension size of 256. It attends over the 8-token CNN voxel encoding and up to 100 additional sequence tokens, with an auto-regressive attention mask. We use a learned positional embedding for each sequence position. An embedding layer lifts each token into an embedding space, consumed by the transformer, and a 2-layer MLP converts Transformer outputs into a probability distribution over tokens.

In all cases we set dropout to 0.1 . We use a learning rate of 0.0005 with the Adam optimizer [2] for all training modes, except for RL, where following CSGNet we use SGD with a learning rate of 0.01 . During supervised pretraining we use a batch size of 400. During PLAD method fine-tuning we use batch size of 100. During RL fine-tuning we use a batch size of 4, due to memory limitations (a batch size of 4 takes up 10GB of GPU memory). Early stopping on the validation set is performed to determine when to end each round and when to stop introducing additional rounds. For deciding when to stop introducing additional rounds, we use a patience of 100 epochs. For deciding when to stop each round, we use a patience of 10 epochs. In both cases we employ a patience threshold of 0.001 IoU improvement (e.g. we must see at least this much improvement to reset the patience). Within each round of PLAD training, we check validation set reconstruction performance with a beam size of 3; between rounds of PLAD training we check validation set reconstruction performance with a beam size of 5; final reconstruction performance of converged models is computed with a beam size of 10.

For RL runs, we make a gradient update after every 10 batches, following CSGNet. For runs that involve VAE training (all Wake-Sleep runs), we add an additional module in-between the encoder and the decoder. This module uses an MLP to convert the output of the encoder into a 128 x 2 latent vector (representing 128 means and standard deviations). This module then samples an 128 dimensional vector from a normal distribution described by the means and standard deviations, and further lifts this encoding into the dimension that the decoder expects with a sequence of linear layers. For each round of VAE training, we allow the VAE to update

P^{BEST} mode	ST	LEST	LEST+ST	LEST+ST+WS
Per round	0.881	1.011	0.853	0.845
All-time	**0.841**	**0.976**	**0.829**	**0.811**

Table 1. Different ways to update P^{BEST} data structure. In the "Per round" row, the data structure is cleared in between rounds. In the "All-time" row, the data structure maintains the best program for each input shape across multiple rounds.

for no more than 100 epochs. We perform early-stopping for VAE training with respect to its loss, where the loss is a combination of reconstruction (cross-entropy on token predictions) and KL divergence, both weighed equally.

2D Experiments For 2DCSG, we follow the network architecture and hyper-parameters of CSGNet. All training regimes use a dropout of 0.2 and a batch size of 100. PLAD methods use the Adam optimizer with a learning rate of 0.001. For deciding when to stop introducing additional rounds, we use a patience of 1000 epochs. For deciding when to stop each round, we use a patience of 10 epochs. In both cases we employ a patience threshold of 0.005 CD improvement. The parameters for the RL runs and VAE training are the same as in the 3D Experiments.

D. P Best Update mode

During updates to P^{BEST}, we choose to update each entry in P^{BEST} according to which inferred program has achieved the best reconstruction similarity with respect to the input shape. The entries of this data structure are maintained across rounds. There is another framing where the entries of this data structure are reset each round, so that the best program for each shape is reset each epoch. This is similar to traditional self-training framing.

We run experiments on 2D CSG with this variant of P^{BEST} update and present results in Table 1. When the best program is maintained across rounds (All-time, bottom row) each fine-tuning strategy reaches a better converged reconstruction accuracy compared with when the best program is reset after each round (Per round, top row).

E. Failure to generalize beyond S^*

As demonstrated by our experiments, PLAD fine-tuning methods are able to successfully specialize $p(\mathbf{z}|\mathbf{x})$ towards a distribution of interest S^*. Unfortunately, this specialization comes at a cost; the fine-tuned $p(\mathbf{z}|\mathbf{x})$ may actually generalize worse to out of distribution samples. To demonstrate this, we collected a small dataset of 2D icons from the The Noun Project[1]. We tested the shape program inference abilities of the initial $p(\mathbf{z}|\mathbf{x})$ trained under supervised pretraining (SP)

[1] https://thenounproject.com

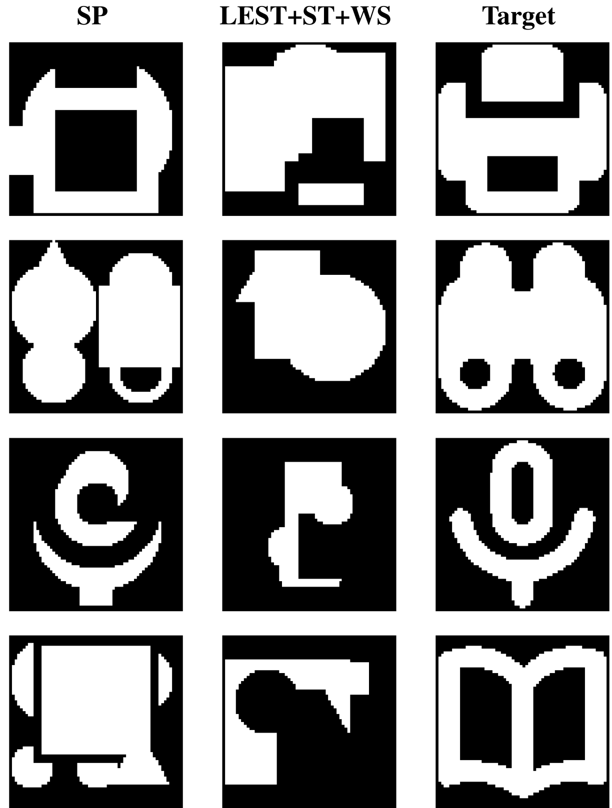

Figure 1. Qualitative examples of inferring 2D CSG programs for 2D icons. Both SP and LEST+ST+WS fail to infer representative programs, but the reconstructions from LEST+ST+WS are even less accurate than those from SP.

and of the fine-tuned $p(\mathbf{z}|\mathbf{x})$ trained under PLAD regimes (LEST+ST+WS) and specialized to CAD shapes. We show qualitative examples of this experiment in Figure 1. While both methods fail to accurately represent the 2D icons, fine-tuning $p(\mathbf{z}|\mathbf{x})$ on CAD shapes lowers the reconstruction accuracy significantly; the SP variant achieves an average CD of 1.9 while the LEST+ST+WS variant achieves a CD of 4.1 Developing $p(\mathbf{z}|\mathbf{x})$ models capable of out-of-domain generalization is an important area of future research.

F. Potential Societal Impacts

Fine-tuning our deep neural networks $p(\mathbf{z}|\mathbf{x})$ requires a relatively large amount of electricity, which can have a significant environmental impact [5]. Reducing the energy consumption of deep learning is an active research area [4,7]. Notably, PLAD techniques place no restrictions on the inference model, making it easy to adopt more efficient deep learning techniques. Moreover, shape program inference procedures may also allow the reverse engineering of protected intellectual property. Thus, improvements in shape program inference may impact the content and enforcement

of copyright law.

G. Additional Qualitative Results

We present additional qualitative results comparing various fine-tuning methods in Figure 2 (2D CSG), Figure 3 (3D CSG) and Figure 4 (ShapeAssembly).

References

[1] R. Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. Shapeassembly: Learning to generate programs for 3d shape structure synthesis. *ACM Transactions on Graphics (TOG), Siggraph Asia 2020*, 39(6):Article 234, 2020. 1

[2] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. 2

[3] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. CSGNet: Neural Shape Parser for Constructive Solid Geometry. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. 1

[4] Ryan Spring and Anshumali Shrivastava. Scalable and sustainable deep learning via randomized hashing. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 445–454, 2017. 3

[5] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019. 3

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. 2

[7] Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Richard G Baraniuk, Zhangyang Wang, and Yingyan Lin. Drawing early-bird tickets: Towards more efficient training of deep networks. *arXiv preprint arXiv:1909.11957*, 2019. 3

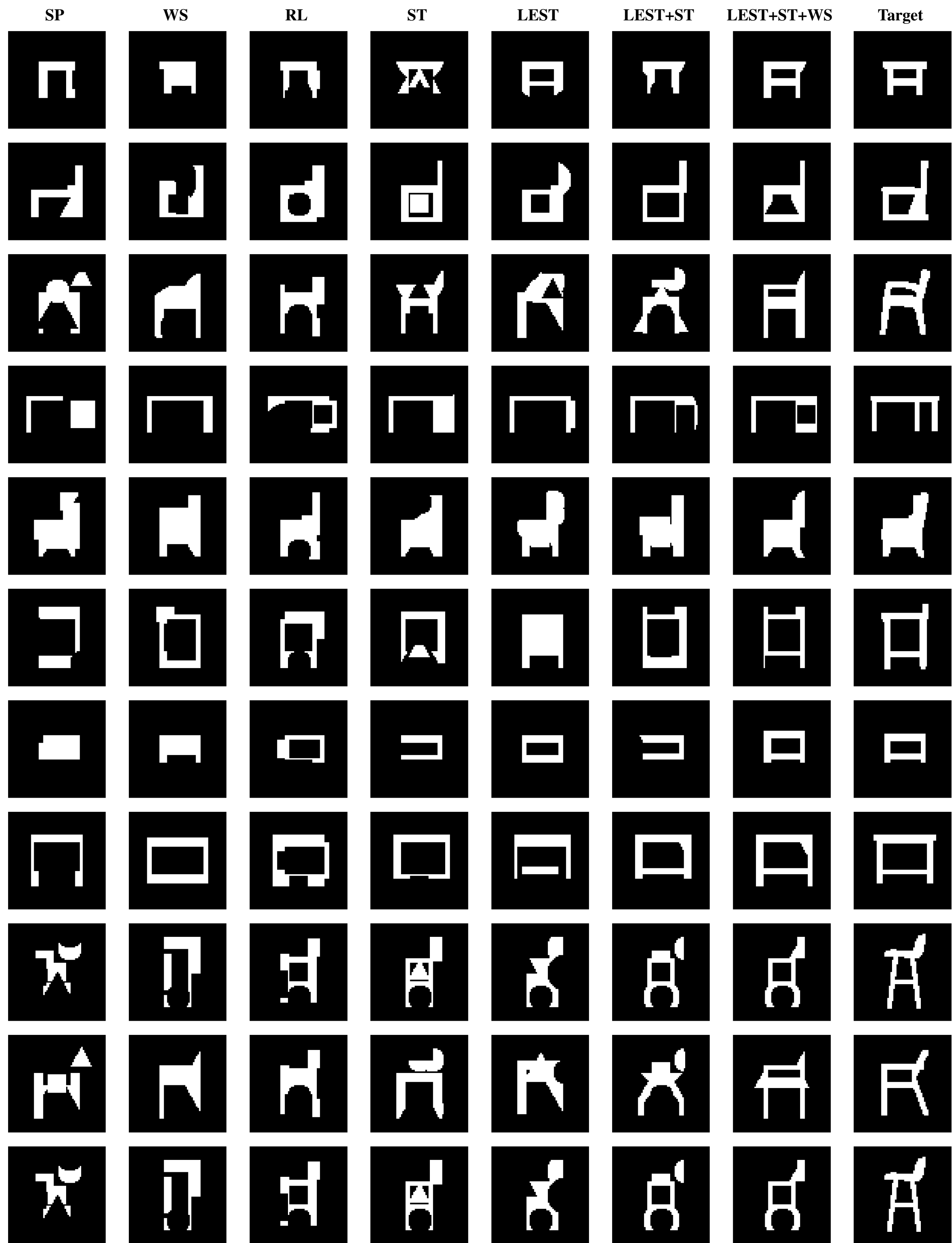

Figure 2. 2DCSG qualitative examples.

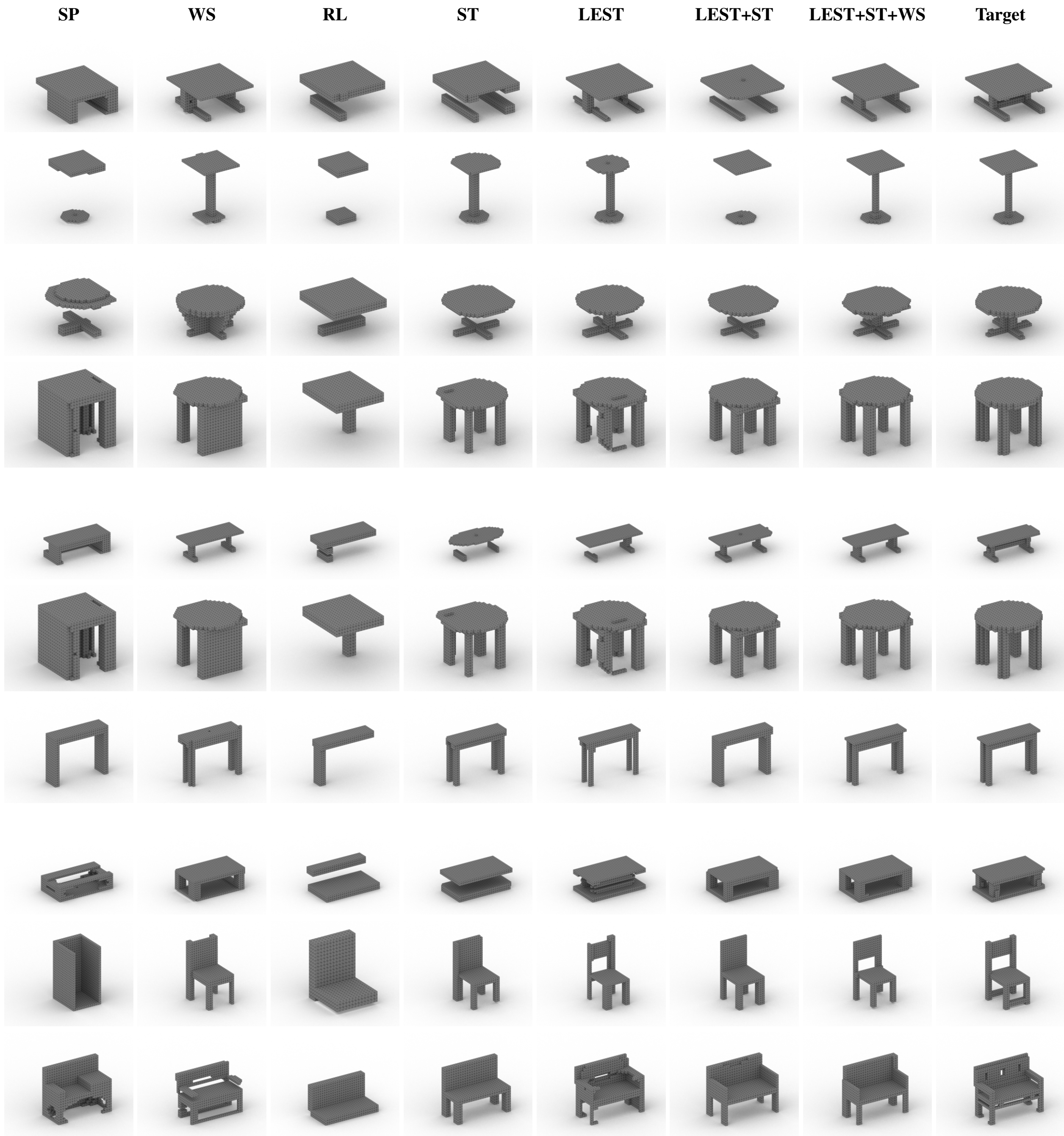

Figure 3. 3DCSG qualitative examples.

Figure 4. ShapeAssembly qualitative examples.