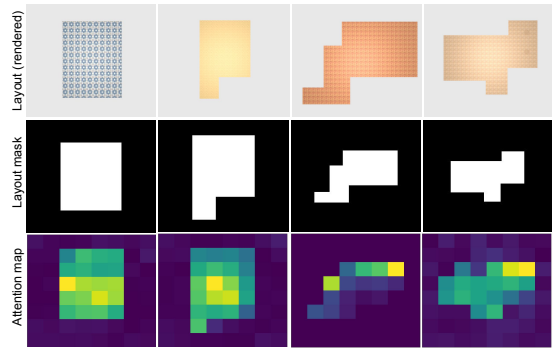
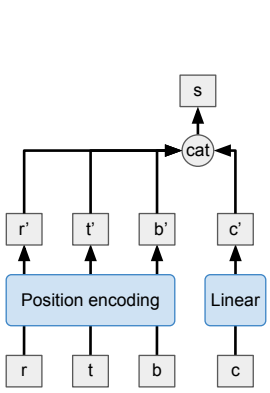


(a) Attention map visualization in the ViT-based layout encoder.

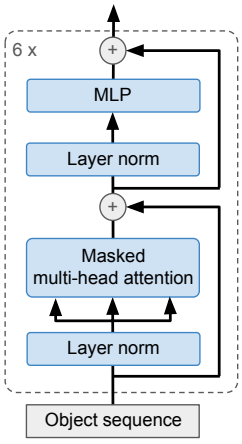
Model	Parameters (M)	KL	FID	CAS (%)
PointNet [51]	**1.13**	8.3	40.5	77.4
ResNet18 [26]	11.1	6.2	36.7	70.1
ViT-S [12]	2.56	**5.9**	**35.2**	**68.0**

(b) Ablation study on different type of layout encoder.

Table 6: Ablation study and visualization on the layout encoder.



(a) Object encoder.



(b) Transformer decoder.

Table 7: Architecture details on the model components.

Transformer decoder. As illustrated in Figure 7b, we take GPT2 [54]-like structure to implement the transformer decoder. Different from the vanilla transformer [64], it uses layer normalization before self-attention block, adds one additional layer normalization after the final self-attention block, and utilizes GeLU as the activation function. Our model configuration employs a smaller variant of GPT-2[3], consisting of 6 layers and a 6-head self-attention mechanism, with a hidden dimension size of 192.

[3] https://github.com/karpathy/nanoGPT

8 Details on evaluation

User study. In our user study, each participant evaluates 204 pairs of images (51 samples $\times$ 4 room types). They are asked to score the quality of the generated indoor scenes based on two criteria: "Which scene fits better within the layout boundary?" and "Which of these indoor scenes appears more realistic?", then select the best.

Tree reconstruction accuracy. We use Average Hierarchical Distance(AHD) to measure the accuracy of reconstructed tree. A scene tree $\mathcal{T}$ can be decomposed into set of node sets $\{S_i\}_{i=1}^{N}$, where N is the total depth of the tree and S_i represents all nodes at the i-th depth. Further, we compare set similarity in average, formally:

$$\text{AHD} = \frac{1}{N}\sum_{i=1}^{N}\frac{|S_i \cap \hat{S}_i|}{|S_i \cup \hat{S}_i|}, \tag{7}$$

where $\hat{S}_i$ is the set at the i-th depth from the ground truth tree. For fair comparison of the different tree reconstruction methods, we select 60 scenes in the experiment and calculate the AHD in average.

Sequence set inconsistency. The score is defined as the Hamming distance of two different sequences in average, which can be formulated as:

$$\text{Inconsistency} = \frac{1}{|N(i,j)|}\sum_{i}\sum_{j\neq i} d_{\text{Hamming}}(s_i, s_j), \tag{8}$$

where s_i is randomly selected sequence and $N(i,j)$ is the number of (i,j) pairs. The score is higher when the the sequences within the set are more inconsistent. Specifically, when the set contains only one sequence, the inconsistency score is zero, as there are no pairs of sequences to compare.

9 Visual results

More visual comparisons. As shown in Figure 10 11 12 13, we compare our results with the-state-of-the-art methods [48, 62] in bedroom and living/dining rooms. Our approach yields more realistic scenes with fewer occurrences of objects extending beyond the layout boundaries.

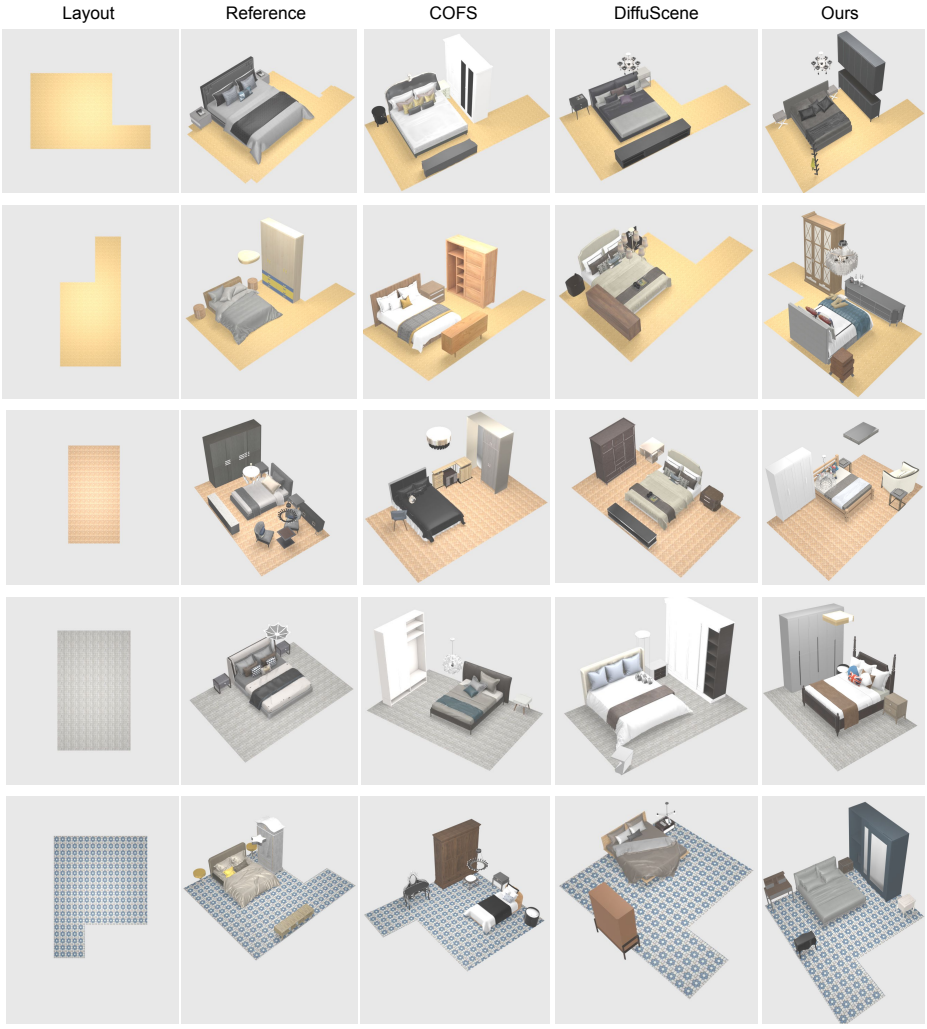


Fig. 10: Scene synthesis. Visual comparison with the-state-of-the-art methods [48, 62] on the bedroom type.

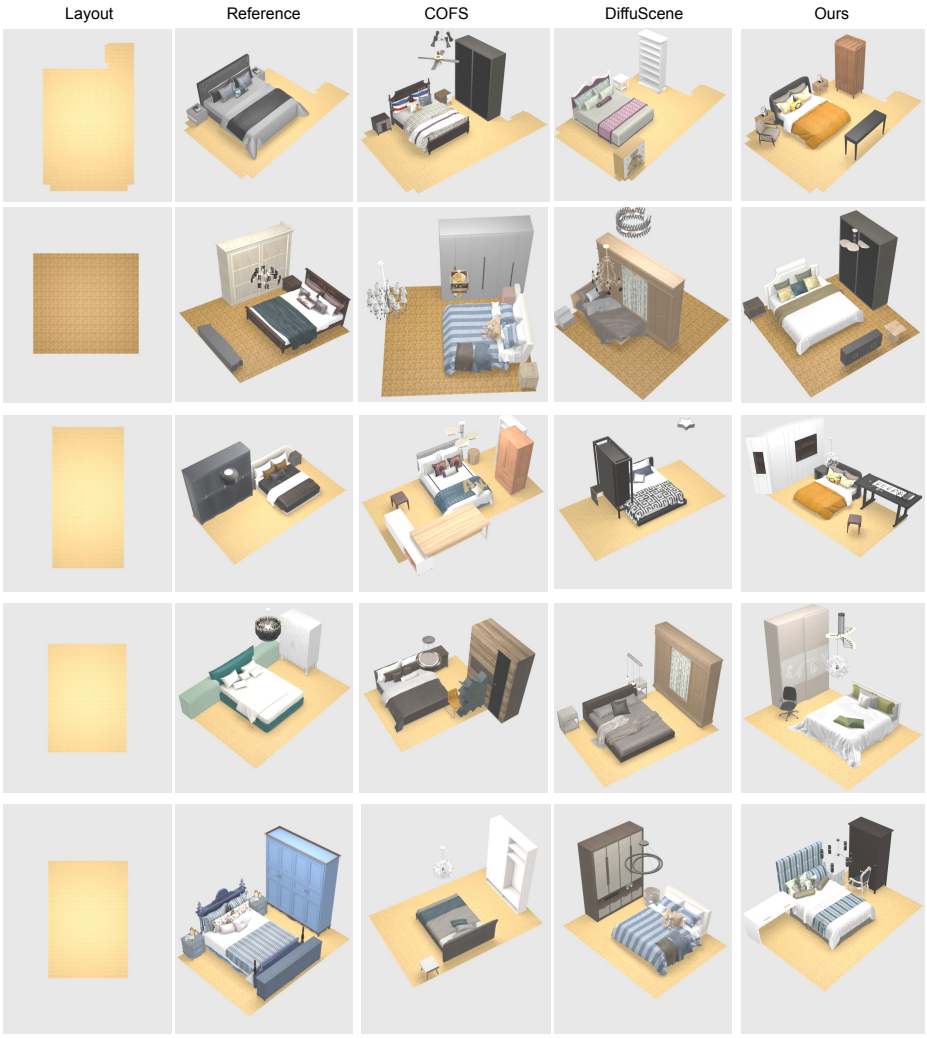


Fig. 11: Scene synthesis. More visual comparison with the-state-of-the-art methods [48, 62] on the bedroom type.

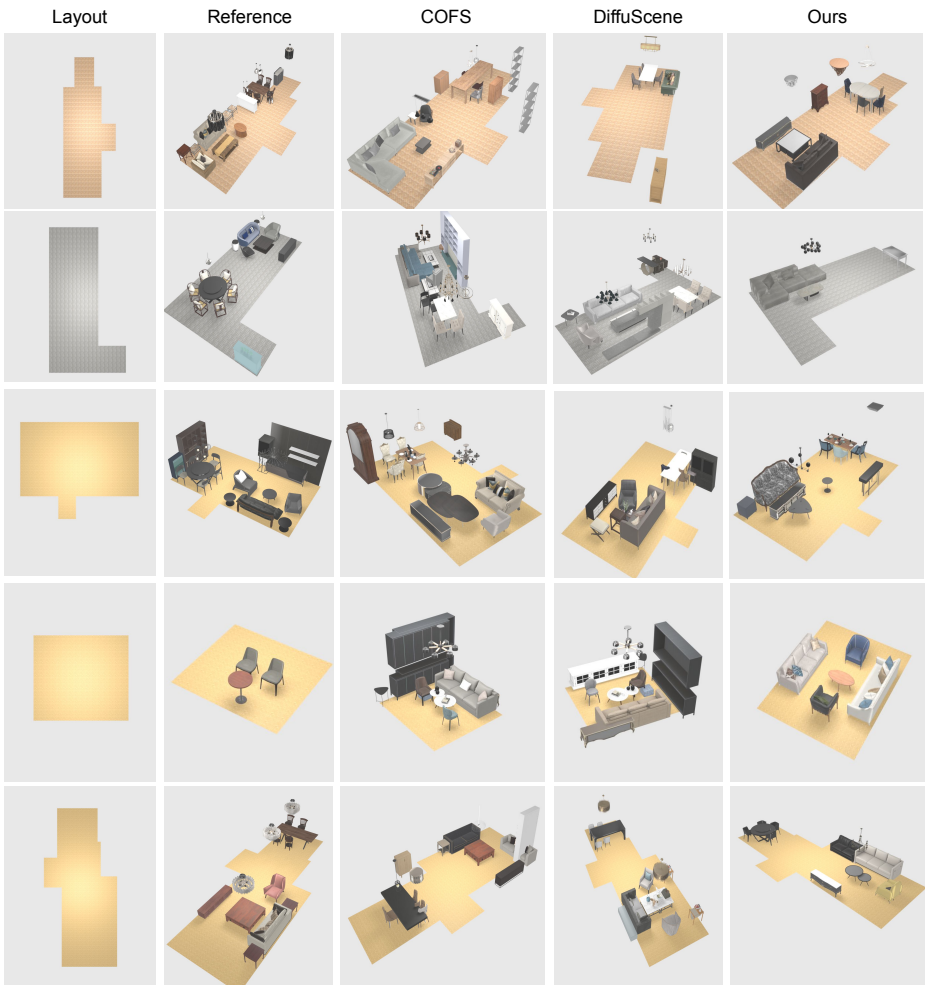


Fig. 12: Scene synthesis. Visual comparison with the-state-of-the-art methods [48, 62] on the living room / dining room type.

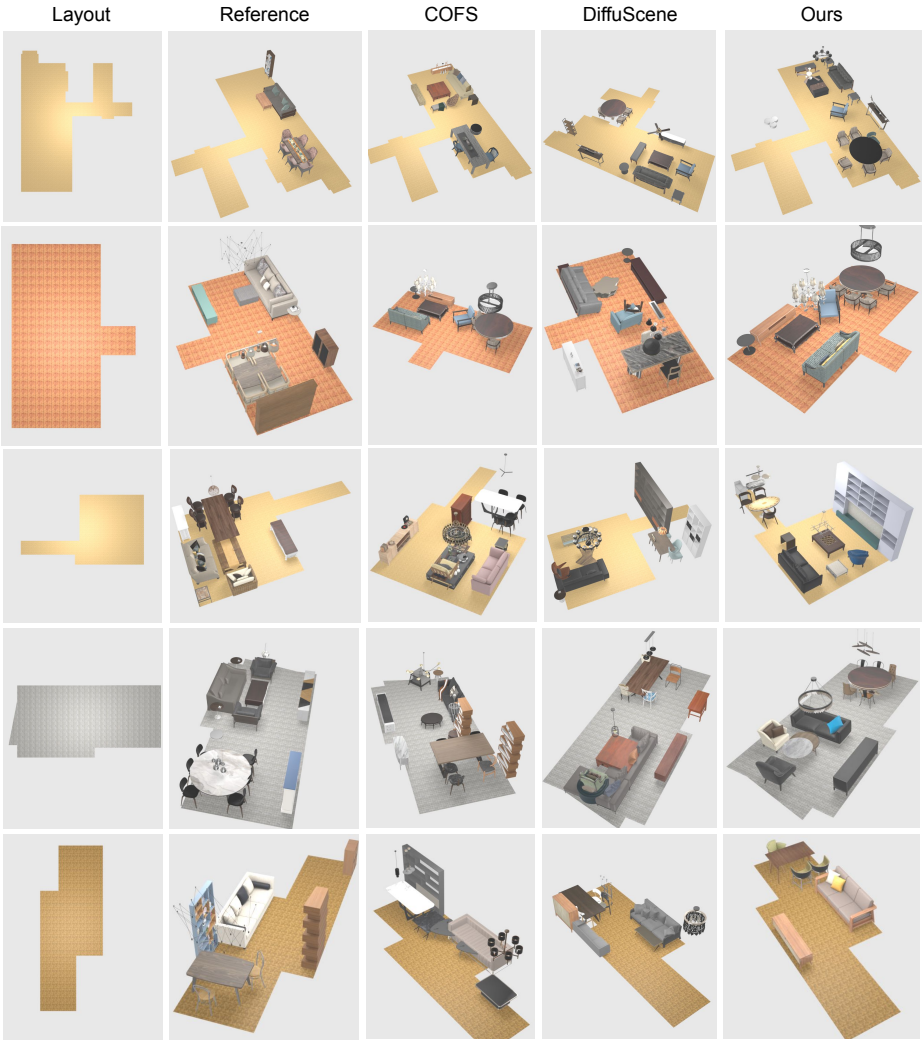


Fig. 13: Scene synthesis. More visual comparison with the-state-of-the-art methods [48, 62] on the living room / dining room type.

Algorithm 1 Pseudocode of Set2Tree algorithm in a PyTorch-like style.

```
def giou(boxes1, boxes2):
    # giou between 2 bboxes
    pass
def find_biggest(subscene):
    # find the biggest furniture in a subscene
    pass
def fn_dist(self, bbox2d_set, xy_set):
    dist_giou = 1 - giou(bbox2d_set) # [N, N]
    dist_enc = sqrt(sum((xy_set[:, newaxis] - xy_set) ** 2, axis=2)) # [N, N]
    # weighted distance matrix
    return dist_euc + k * dist_giou

"""
Definition of Furniture: node of Tree
"""
class Furniture:
    def __init__(self, x1y1x2y2=None, xy=None, feature=None):
        self.bbox2d = x1y1x2y2
        self.xy = xy
        self.feature = feature
        self.children = []
    def add_node(self, furniture):
        self.children.append(furniture)

"""
Definition of Tree
"""
class Tree:
    def __init__(self):
        self.root = Furniture() # empty
    def shuffle(self):
        pass # easy to implement
    def bfs(self):
        pass # easy to implement
    def set2tree(self, f_set, labels):
        # organize the furniture set into a scene tree
        for l in len(set(labels)):
            if l != -1: # not outlier
                subscene = [f_set[j] for i in labels if i == l]
                # find the largest furniture as parent node
                f_parent = find_biggest(subscene)
                # add the parent node under the root node
                self.root.add_node(f_parent)
                # add the remaining objects under the parent node
                for f in subscene:
                    if f != f_parent:
                        f_parent.add_node(f)
```

`iou`: intersection-over-union of two bounding boxes; `sqrt`: square-root; `sum`: summarization; `newaxis`: new axis of an array.

Algorithm 2 Pseudocode of Tree2Forest algorithm in a PyTorch-like style.

```
from sklearn.cluster import DBSCAN
from numpy import array
from copy import deepcopy
def tree2forest(f_set, eps=0.15, min_samples=2):
    """
    Prepare the distance matrix
    """
    bbox2d_set = array([f.bbox2d for f in f_set]) # [N, 4]
    xy_set = array([f.xy for f in f_set]) # [N, 2]
    dist_mat = fn_dist(bbox2d_set, xy_set)
    """
    DBSCAN clustering
    """
    dbscan = DBSCAN(eps, min_samples, metric='precomputed')
    labels = dbscan.fit_predict(dist_mat)
    tree = Tree() # empty tree
    tree.set2tree(f_set, labels) # build tree
    forest = [] # create empty forest
    # find all outliner
    outliers = [f_set[j] for j, k in enumerate(labels) if k == -1]
    for f in outliers: # process each outlier
        # add to the parent nodes
        for i in range(len(tree.root.children)):
            tmp_tree = deepcopy(tree)
            tmp_tree.children[i].add_node(f)
            forest.append(tmp_tree)
        # add to the root node
        tmp_tree = deepcopy(tree)
        tmp_tree.root.add_node(f)
        forest.append(tmp_tree)
    return forest
```

Algorithm 3 Pseudocode of Forest2Seq algorithm in a PyTorch-like style.

```
"""
Flatten a forest (set of tree) to a sequence
"""
def forest2seq(forest):
    """
    forest: list of Tree
    return: list of Furniture
    """
    N = len(forest) # number of trees
    idx = randint(0, N) # randomly select one tree
    tree_selected = forest[idx]
    return tree_selected.shuffle().bfs()
```

`bfs`: breadth-first traversal; `shuffle`: tree shuffle; `randint`: return random integers from low (inclusive) to high (exclusive).