COFS
CONTROLLABLE FURNITURE LAYOUT SYNTHESIS SUPPLEMENTARY MATERIAL

Anonymous authors
Paper under double-blind review

ABSTRACT

In this supplementary document accompanying our main submission, we first include a discussion contextualizing COFS in relation to concurrent and previous scene generation methods. We then present some generation results. Following that, we describe our architecture and experimental setups in greater detail. In particular, we describe each of the components of our architecture, including the training protocol, our metrics, and the design of the user study. Then we provide more details on the sampling strategy that we employ, followed by a comparison of layout generation times and parameter counts to existing methods. Additionally, we perform an ablation study justifying our design choices. We conclude with additional qualitative results and a table of key notation used in the main paper.

A DISCUSSION

Q: *What exactly is the advantage of the proposed method over ATISS?*
A: ATISS permutes whole objects during training which makes it invariant to the order of objects. However, by design, ATISS has a specific sampling order of attributes (cf. Section H and Equation 5). This leads to certain undesirable attributes:

1. One does not get all conditional distributions of attributes, but only a few. As an example, one cannot get the probability of an object given a location - $p(\tau|t)$ but one can only compute the probability of a given class being at some location - $p(t|\tau)$.

2. A lack of these probabilities means one cannot perform usable conditioning. During sampling, one cannot ask a single trained model to perform both these tasks - 'Given a location, what is the most likely class here?' and 'Given this class, where is the most likely location?'.

The kinds of conditioning described in (2) regularly arise in the course of game development. One often needs a large amount quickly generated layouts with some control over the generated layout - in some layouts, one might only need to specify the class of an object. For example, a lever or an item that triggers an action. In others, we might need to specify more attributes - if one wants to generate office scenes on a large scale, there would be a lot of constraints on the angles of chairs and desks (with chairs facing desks), and chairs in offices often facing the door. While designers can generate a few examples in reasonable time, generating a few dozen or so exploratory layouts can often be tedious.

ATISS is incapable of performing this form of conditioning. In COFS, on the other hand, any subset of the attributes may be masked, which lets the network infer that attribute, or unmasked which makes the attribute a constraint. This allows for more flexible forms of conditioning. We show a few examples of conditioning on angles and classes in Fig. 2.

This also allows to us train only a single model for *all* modes of conditioning. Furthermore, our network has fewer parameters and is faster to sample from (cf. Table 1).

Q: *But ATISS shows results on* Object Suggestion *at a given location.*
A: It is true that ATISS shows results showing generation of objects at a given location. However, they perform the task in the following way (Paschalidou et al., 2021):

> ... We now test the ability of our model to provide object suggestions given a scene and user specified location constraints. To perform this task we sample objects from our generative model and accept the ones that fullfill the constraints provided by the user...

In practice, this means ATISS performs rejection sampling hoping that randomly some object lies withing the provided constraints. In their code, this means they sample a 100 samples before giving up. Based on our timing results (cf. Table 1), this is significantly slower than our method by a few orders of magnitude.

For our COFS results with constrained locations, we simply set location/translation attribute to the desired value ($t = t_0$) on the encoder side and keep it unmasked. Then we only have to sample once, as the distribution being sampled from on the decoder-side is the distribution $p(c, t = t_0, e, r)$. This makes fine-grained conditioning more practical.

Q: *What is the practical use-case for having complex constraints?*
A: As answered in the first question, one practical use is in the game-development industry where one needs a large number of layouts with a few constraints on the classes, locations, or their combinations.

Another scenario is interior-design where clients often want *something* to put in a location (implying constraints on location, and inferring object type and other attributes). Clients oftentimes also want an object *somewhere* so that it does not *block* some other object. This implies a constraint on both location and spatial extent. Rotation constraints might also be required, when clients want a dominating direction - for example constraining the rotation of a dining table or couch/sofa, which in turn sets constraints on how the other furniture can be placed.

Q: *What are the different types of position tokens?*
A: In our early experiments, we tried a MaskGit (Chang et al., 2022) style decoder-only architecture. However, MaskGit is trained with absolute position tokens, a design decision, which makes sense in an image generation setting - when generating faces, one expects eyes to always be above the nose. But using absolute position encodings breaks permutation invariance.

So we trained a model with no positional tokens at all. However, this model performed very poorly on both conditional and unconditional generation. There are two reasons:

- Without some additional tokens specifying which attributes belong to and define a single object, the normal attention mechanism treats all tokens as the same. While what we want is that attributes of an object influence its other attributes more. This problem is present even during training as there is no way of letting the network *know* that the 'next 5' tokens belong to a single object as there is no concept of the *next or previous.*
- Without absolute position encodings, sampling is incredibly difficult. This has been shown in multiple examples in NLP literature (Vaswani et al., 2017; Lewis et al., 2020; Wang & Cho, 2019; Radford et al., 2019).

We then added tokens which let us specify a single object - the Object Index Token, $\mathcal{O}^i$ which is added to embeddings of all attributes that define a single object. This helps the model disambiguate between tokens that belong to different objects. This model worked well for tasks like outlier detection but still performed poorly in tasks which needed inference/generation of more tokens like unconditional or unconditional generation. This makes intuitive sense as for outlier detection, the network, with Object Index Token can disambiguate between different objects and can tell when an object is *wrong*.

However, the tokens within an object are still treated the same, so we added the Relative Position Token $\mathcal{R}^i$. This token is shared across similar attribute types. Within each object, the Relative Position Token demarcates what the token represents. We have one token corresponding to each attribute type - τ, t, e, r (cf. Table 2).

With these two additional tokens, the performance was better but still not the same quality as ATISS, especially in unconditional generation. We hypothesized that is in part because Non-Autoregressive Sampling still does not achieve the same quality as Autoregressive Sampling.

Hence, we came up with the architecture described in the main paper - where the encoder has only $\mathcal{O}^i$ and $\mathcal{R}^i$. This makes the encoding a set-based encoding as there is no notion of order. And we can still

get very high quality samples, as the decoder side which produces the samples is still autoregressive, with the usual Absolute Posiition Token $\mathcal{P}^i$.

This final model separates the duties of conditioning and generation between the encoder and the decoder, with the encoder performing the conditioning and the decoder the generation. The encoding is set-like, which allows for many forms of fine-grained conditioning. The decoding is the usual sequence decoding which leads to high-quality samples.

See also the ablation and discussion in Sec. E.1 about how using both encodings on the encoder-side helps the decoder decide where the attributes must be sampled.

B Additional Generation Results

B.1 Qualitative Conditional Generation

Location-conditioned generation: In this section, we show some qualitative results from our model that other methods cannot generate. In particular, we show in Fig. 1 how to perform location-conditioned generation. In order to perform this, we use an empty scene and encode its boundary representation $\mathcal{I}$ with our image encoder. Then, we fix the locations in the sequence C and allow the model to sample the distribution for the classes. This is useful as a *suggestion* module which can be used to provide suggestions to a user using the system. The results show that COFS learns the distribution of layouts with `nightstands` being close to the room edges on either sides of the beds. Additionally, we see `tv-stands` close to the center of the rooms aligned with the locations of beds.

For ease of visualization, instead of showing the whole distribution over the possible classes, we choose to show the most-likely class.

More fine-grained generation: In this section, we provide more examples of the kinds of conditioning COFS supports. This is used as a demonstration of the use-case suggested in the discussion - we have a few constraints on the angles and we wish to generate different layouts under different constraints. This is shown in Fig. 2 We generate a whole layout from scratch, but constrain the angles of beds. As seen in the results, the rest of the layout adjusts to satisfy the imposed constraint. We further impose few more constraints - that there should exist a `dressing-table` and it should be oriented opposite the bed. COFS handles this constraint well with the model learning to change one of the otherwise generated objects (mostly `tv-stand` or `wardrobe`) into a `dressing-table`.

Distributions under fine-grained conditioning: We now show how the encoder allows COFS to *look-ahead*. In Fig. 3, we show the distributions for two classes - `nightstand` and `tv-stand`. In general, `nightstand` are to either sides of the `beds`. And the `tv-stand` opposite the `bed`. To generate the distributions in COFS, we start with an empty layout and set the constraint sequence C, as follows - object 2 class is set to be `bed` and it's location is given. We then sample from the model autoregressively but setting the object 1 class to be either `nightstand` or `tv-stand`. This shows the idea behind having an encoder. With the encoder, we can introduce conditions that occur in the *future*. This form of conditioning is otherwise impossible in autoregressive models, as they need to respect causality. We see this uncertainity particularly in the `tv-stand` class where the ATISS model is certain that the class is towards the edges - as seen by two peaks near the floorplan boundary, but the model is not sure exactly which boundary. On the other hand, COFS, which is conditioned on the *future* location of the `bed` can ignore the edge where the `bed` itself is, as a `tv-stand` exists opposite to the `bed`.

However, while this form of conditioning works well in general, there is no guarantee that the constraints will be satisfied. For example, in the L-shaped floorplan (highlighted in red), the distributions are already very sharp. This is in part because the 3D-FRONT dataset lays out L-shaped floorplans in a very specific manner - mostly the bed lying on the same side as the L. This also leads to failure cases where the model ignores the provided condition, because of the strong prior from the floorplan boundary.

B.2 Qualitative Unconditional Generation

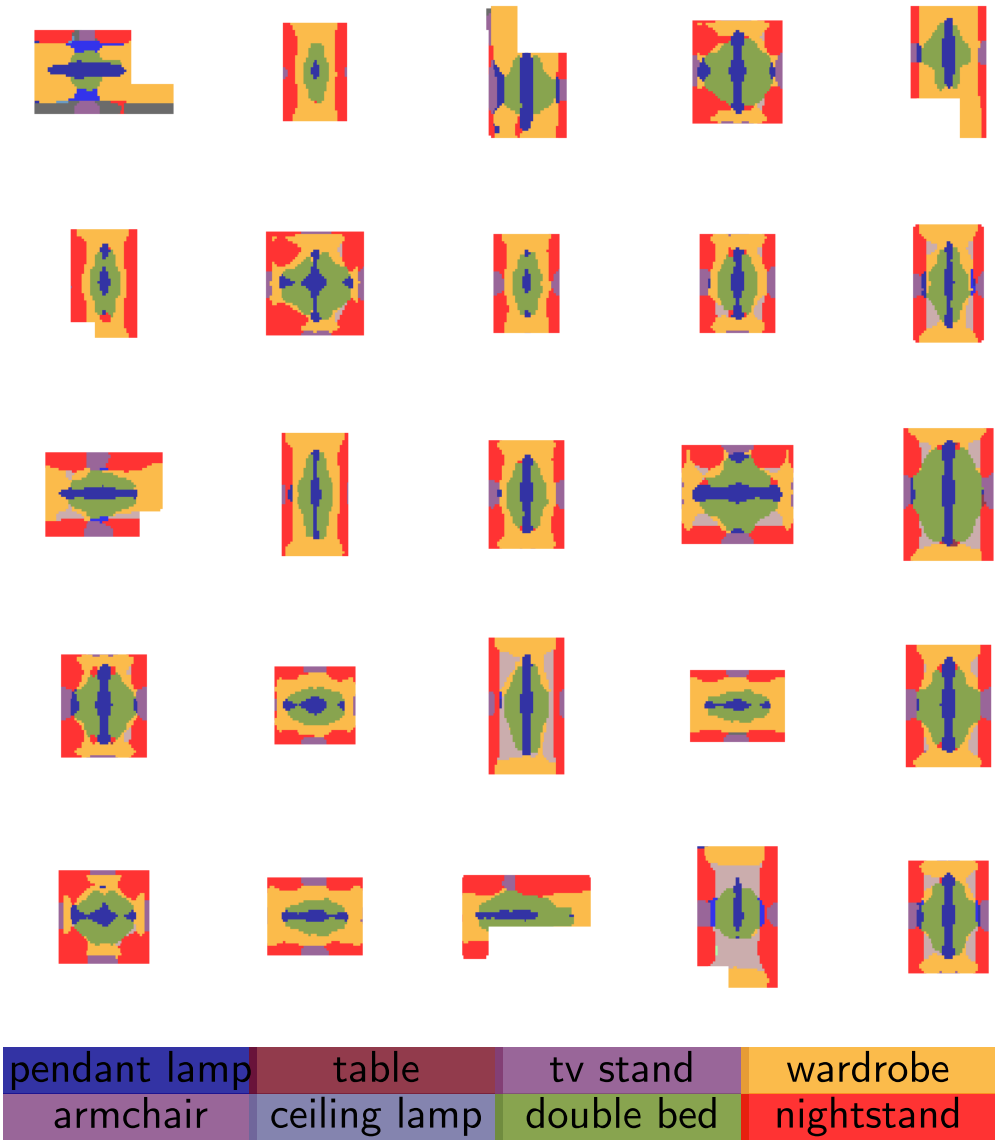


Figure 1: **Attribute-level conditioning:** For different layouts, we use the location as a constraint, and let the model infer the classes. This is only possible because of our encoder based architecture. Our model predicts classes that suit the location. ATISS on the other hand, cannot be conditioned in such a manner.

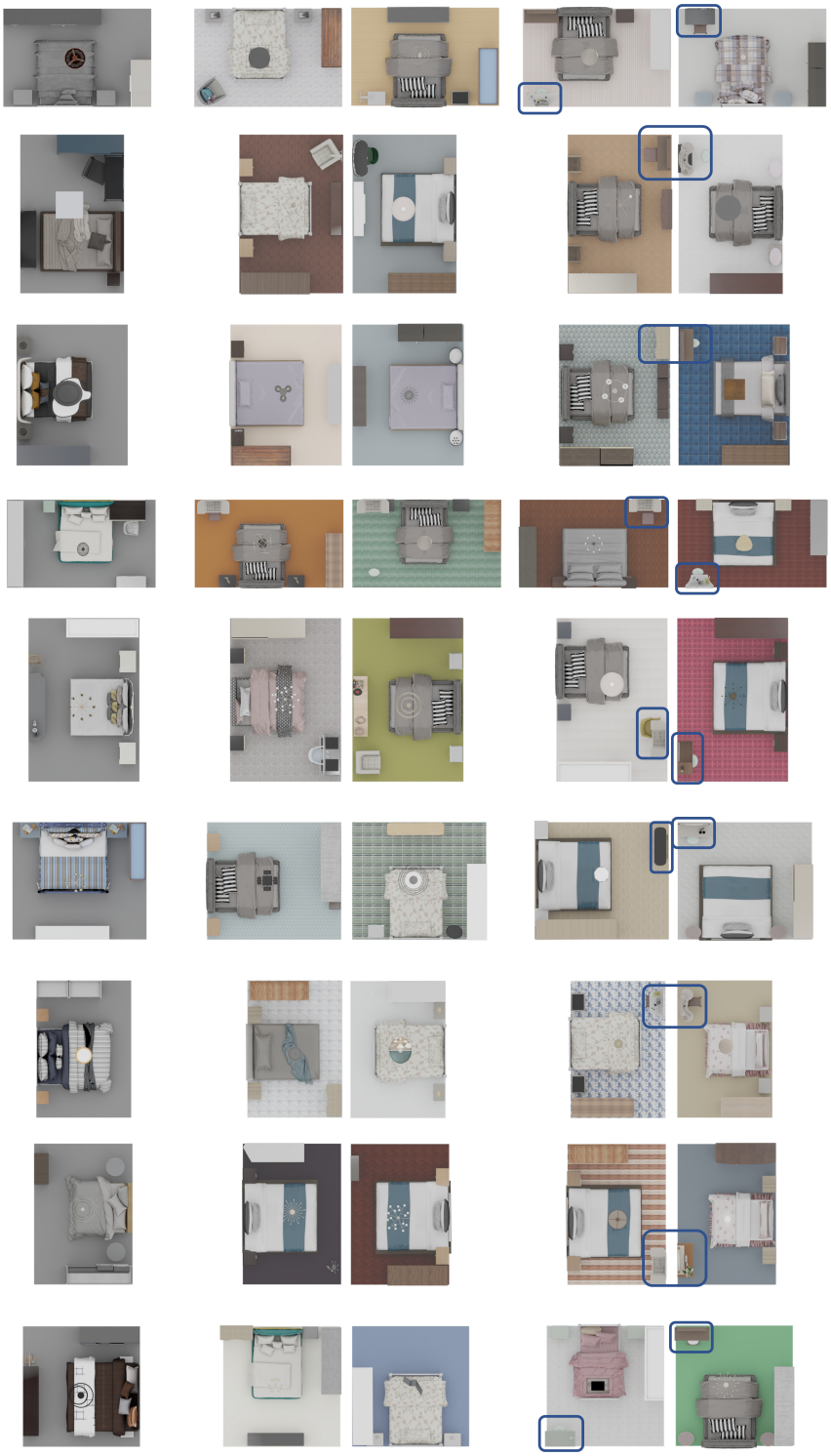


Figure 2: **Fine-grained conditioning:** GT is on the left. In the middle two columns, we constrain the the first class to be `bed` with two angles, and see that the rest of the layout arranges to fit the constraints. In the next two columns, we constrain the layout to contain a `dressing-table` at an angle opposite to the `bed`. The `dressing-tables` are highlighted in the blue boxes.

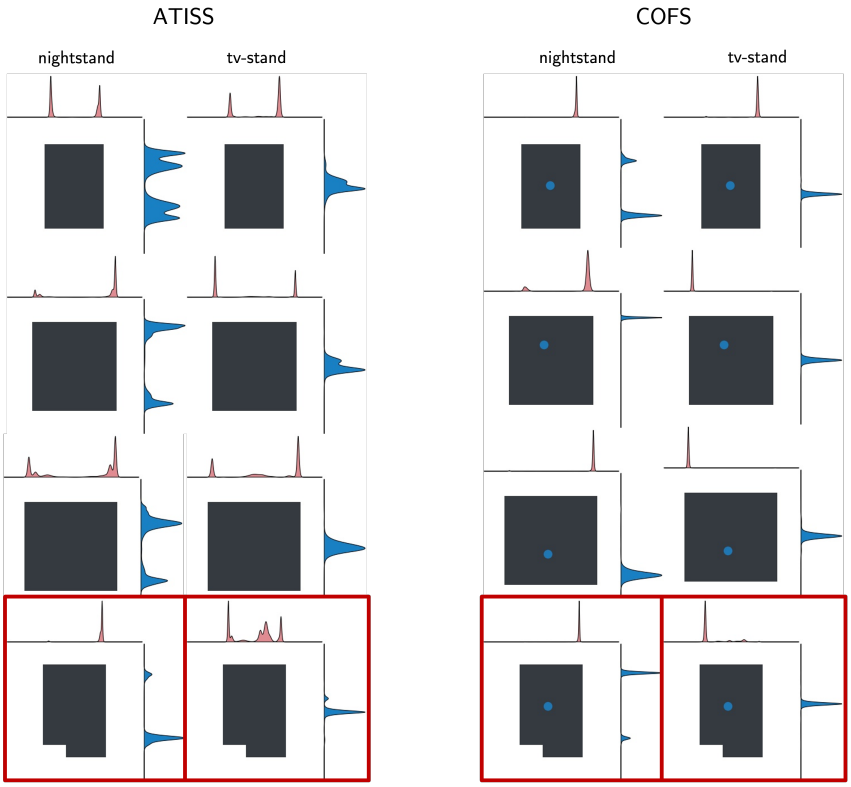


Figure 3: **Look-ahead:** ATISS cannot condition on tokens yet not generated, hence it has distributions that are flat (uncertain) about the positions of the classes. In contrast, COFS can **look-ahead**, hence the probabilities are much sharper as the locations of the generated classes are more constrained by the location of the `bed` - shown by the blue circle.

We show some qualitative examples of unconditional generation in greater detail.

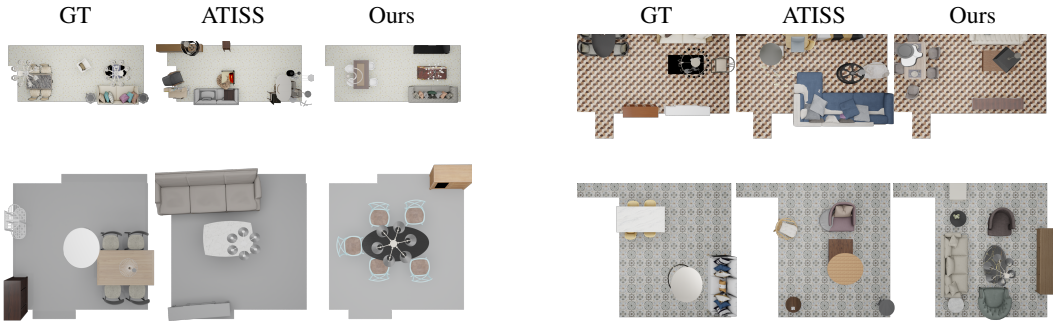


Figure 4: **Unconditional generation.** We compare generated scenes from GT, ATISS, and COFS. Both ATISS and COFS are conditioned on the floorplan boundary (GT). In contrast to ATISS, we can see that our model consistently creates plausible layouts within the floorplan boundary while avoiding unnatural object intersections. These are results on the challenging LIVING (column 1) and DINING (column 2) categories.(Best viewed zoomed in, on a computer display)

C DETAILS ON ARCHITECTURE AND EXPERIMENT SETUPS

We base our architecture on ATISS (Paschalidou et al., 2021) in order to ensure a fair comparison to our closest competitor, using the same underlying library (Katharopoulos et al., 2020). Consequently, most of the building blocks are shared. However, we would like to describe our architecture in greater detail in this section for the purposes of reproducibility.

Hyperparameters: We implement our models in PyTorch 1.7.0 (Paszke et al., 2019) . We use standard transformer blocks for the encoder and decoder except that the ReLU activation is replaced with GeLU (Hendrycks & Gimpel, 2016). We use 4 encoder layers and 4 decoder layers, with a hidden dimension of 256 and 4 attention heads yielding a query vector of dimension 64. We use a batch size of 128 sequences and train on a single nVIDIA A100 GPU with the AdamW (Loshchilov & Hutter, 2019) optimizer which we found to be more stable than Adam (Kingma & Ba, 2015). We use weight decay of 0.001 and clip the gradient norm to be a maximium of 30. We found that the networks begins to overfit very early, especially for classes other than BEDROOM, because of the scarcity of data. Thus, for training networks on other classes, we pre-train on the BEDROOM class, and then reuse those weights as initialization. We do not use any form of learning rate scheduling as our experiments did not suggest significant performance gains. We train for 1000 epochs and use early stopping.

Layout sequence: During training, we construct the sequence S corresponding to the layout by arranging the object bounding boxes in a random order with a permutation π and concatenating their bounding box attributes as individual tokens.

$$\begin{aligned} &s_2 = \tau_{\pi_1},\ s_3 = (\tau_{\pi_1})_x,\ s_4 = (\tau_{\pi_1})_y,\ s_5 = (\tau_{\pi_1})_z,\ s_6 = (e_{\pi_1})_x,\ \cdots\ s_9 = \tau_{\pi_2}\ \cdots \\ &s_1 = \texttt{SOS},\ s_k = \texttt{EOS} \end{aligned} \tag{1}$$

where τ_{π_1} and $(t_{\pi_1})_x$ represent the class and $x-$translation of the first object after permutation, τ_{π_2} represents the class of the second object after permutation and so on.

Object attributes are always flattened the same way in our implementation, although in principle the attribute order can itself be permuted. We use the same attribute order for ease of implementation.

Embeddings: We described how we generate embeddings for the tokens in C and S. We use a learnable matrix E_{class} of dimension $n_\tau \times 256$ to encode the type τ_i, with each row corresponding to one class. We use an additional `[MASK]` class. For the other attributes of translation (t_i), size (e_i) and rotation (r_i), we use sinuosoidal positional encodings (Vaswani et al., 2017; Paschalidou et al., 2021) with 128 levels ($L = 128$). We call these embeddings γ:

$$\gamma(b) = \begin{cases} (\sin(2^0\pi b), \cos(2^0\pi b), \ldots, \sin(2^{L-1}\pi b), \cos(2^{L-1}\pi b)) & \text{if} \quad b \in \{t, e, r\} \\ E_{class}[\tau, :] & \text{if} \quad b \in \{\tau\} \end{cases} \tag{2}$$

For the encoder, the embeddings of $\mathcal{R}^i$ are a learned matrix E_r of dimension 8×256. Each row corresponds to a different type of attribute - one for type, 3 each for translation and size, and one for

the rotation. The embedding of $\mathcal{O}^i$ are again a learned matrix E_o of dimension $k \times 256$, where k is the maximum number of objects. For the decoder, the embeddings of $\mathcal{P}^i$ are also a learned matrix E_p of size $n \times 256$. The final embeddings are the *sum* of the corresponding embeddings:

$$\begin{aligned} \gamma_e(b_i) &= \gamma(b_i) + E_r[\mathcal{R}^i, :] + E_o[\mathcal{O}^i, :] \\ \gamma_d(b_i) &= \gamma(b_i) + E_p[\mathcal{P}^i, :] \end{aligned} \tag{3}$$

where γ_e and γ_d are the encoder and decoder embeddings respectively.

Optimizer: We use the PyTorch implementation of the AdamW optimizer with the default parameters for our model with a constant learning rate of 10^{-4} and weight decay set to 10^{-3}. We linearly warmup the learning rate for 2000 steps. In addition, we found gradient clipping[1] to be necessary to ensure convergence. We set the maximum gradient norm to be 30. Empirically, we found that setting the gradient norm to be low led to slower convergence.

We train with a batch size of 128, and train for 1000 epochs. We perform validation every 5 epochs. We save the model with the best performance on the validation set. We use random rotation augmentation by randomly rotating each scene between 0 and 360 degrees.

We wish to clarify that while we used the AdamW optimizer for our model, we used the vanilla Adam optimizer for ATISS, as described in (Paschalidou et al., 2021).

Parameter Probability Distributions: We need to predict object attributes from the final transformer decoder outputs. To this end, we use use MLPs to go from the embedding dimension to the parameters of the distribution describing the attributes. For the class τ, we use a linear layer from the embedding dimension to the number of classes. For the other attributes, we use MLPs with one-input layer $(256, 512)$, one hidden-layer $(512, 256)$, and one output-layer $(256, 30)$ and ReLU activations. The output size reflects that we use a mixture distribution with 10 components, and each component-distribution is parameterized by 3 values.

Transfer Learning: The datasets LIVING, DINING, LIBRARY are much smaller compared to the BEDROOM dataset. Thus, we use a transfer learning approach, where we first train on the BEDROOM dataset, and use those weights as an initialization, when training on the smaller datasets. This reduces the training time significantly, as well as combats overfitting on the smaller datasets.

We note that the datasets have a slightly different number of classes, thus any weights associated with the number of classes are not transferred, but instead sampled from a Normal Distribution, with mean 0 and standard deviation 0.01.

C.1 METRICS FOR UNCONDITIONAL GENERATION

The evaluation protocol follows ATISS closely, but we describe it here for the sake of completeness.

To compute the *KL-divergence*, we simply create a histogram of object categories in the generated layouts g_i and the ground-truth gt_i, where $1 \leq i \leq n_{class}$ and use the the formula for the categorical KL-Divergence:

$$KL(gt_i \| g_i) = \sum_i gt_i \log \left(\frac{gt_i + \epsilon}{g_i + \epsilon} \right) \tag{4}$$

where $\epsilon = 10^{-6}$ is a small constant for numerical stability.

To compute the *Classifier Accuracy Score* (CAS), we use an AlexNet (Krizhevsky, 2014) [2] model pretrained on ImageNet (Deng et al., 2009) to classify the orthographic renderings as real or fake.

To compute the *FID*, we render both the ground-truth and generated layouts from a top-down view into a 256×256 images with an orthographic camera using Blender v3.1.0 (Community, 2018). Following ATISS, the FID is computed using the code from Parmar et al. (Parmar et al., 2022) [3] We will release the `.blend`-file used for rendering upon acceptance.

[1] We use `torch.nn.utils.clip_grad_norm_`

[2] `torchvision.models.alexnet` Weights

[3] https://github.com/GaParmar/clean-fids, commit `fca6718`

C.2 Additional Details on the Perceptual Study

We conducted a user study to establish the quality improvement provided by our method over ATISS (Paschalidou et al., 2021). For this, we used all the 224 scenes from the test set of BEDROOM and sampled layouts for the ground-truth floorplan boundary, with both ATISS and COFS. The samples were generated with no post-processing except object-retrieval.

Each user was presented the layouts generated by both methods in a side-by-side comparison. The viewer was completely interactive allowing the user to pan, scroll and zoom. Following ATISS, we added specific instructions to the user to ignore the texture of the furniture in the layout and instead focus on the arrangement of objects. The location (left/right) where the ATISS/COFS layout was displayed was randomized to avoid bias.

Figure 5: **User-study interface:** We presented participants with a browser based interface that allowed interactivity with the generated layouts.

This setup was created to mimic the user study conducted in ATISS with the added availability of an interactive interface where users could rotate the scene, pand, and zoom in/out, instead of merely having a rotating GIF. This interface was repeated for both the unconditional generation and the attribute-conditioned generation settings. We received a total of 326 responses.

We reported the results in the main submission, for both realism - *"amongst the two methods, which produced more realistic furniture layouts?"* and error - *"for each of the methods, which method had errors?"*. Instead of error, we plot $(1 - error)$ so that small values become clearer. We found that users overwhelmingly preferred COFS generated layouts to ATISS layouts in both the unconditional and the attribute-conditioned settings, with stronger preference (about 3% higher) in the attribute-conditioned setting showcasing the strength of our method in fine-grained conditioning. Furthermore, COFS had a very low error rate of approximately 4% in the attribute-conditioned setting while the error rate for ATISS was $4\times$ higher at roughly 16%, again highlighting that our method produces realistic layouts that can satisfy the input constraints/conditions.

There was no worker compensation involved for the participants of the study.

D 3D-Front Dataset

To the best of our knowledge, the 3D-Front (Fu et al., 2021) dataset is the largest collection of indoor furniture layouts in the public domain. Its large scale is obtained, in part, by employing a semi-automatic pipeline, where a machine-learning system places the objects roughly, and an optimization step (Weiss et al., 2019) refines the layouts further to conform to design standards. The only human involvement is verification that the layouts are valid - do not have object intersections,

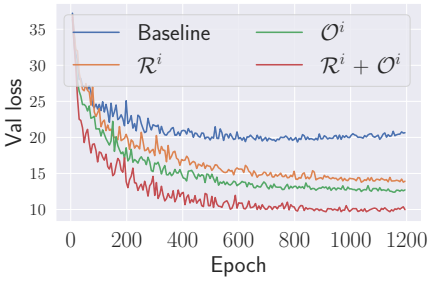
Figure 6: We show a few examples of inconsistencies in the 3D-FRONT dataset. **Top:** Camera placement in 3D-Front layouts. **Center:** The corresponding regions show errors in the ground-truth data. *Left:* Chairs facing and intersecting a shelf. *Right:* Chairs in the correct orientation, but intersecting with a table. **Bottom:** Some more ground-truth errors. *(From Left to Right:)* Intersection. Blocking. Wrong Orientation and Intersection. Wrong Orientation.

objects that block doors, etc. However, in our exploration, we find several inconsistencies still remain in the dataset. We mention a few - nightstands intersecting their nearest beds, nightstands obstructing wardrobes, chairs intersecting their closest tables, and chairs that face in the *wrong* directions. We point out a few of these examples in Fig. 6.

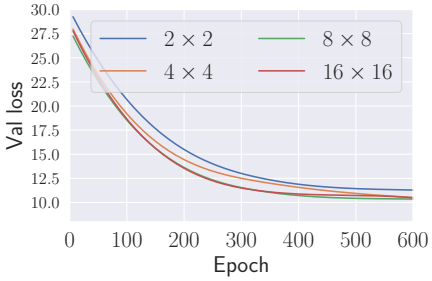
Our method, like other data-driven methods, learns the placement of objects from data. Thus, any errors in the ground-truth data itself would also show up in the sampled layouts. This is true, especially for BEDROOM dataset, where the sampled nightstands often end up intersecting with beds.

E ABLATIONS

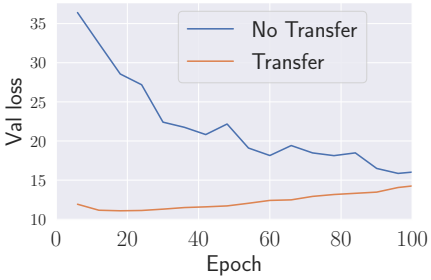
In this section, we justify our design choices by conducting an ablation study. We train our model under different settings on the BEDROOM dataset, unless specified otherwise, and use the validation loss, the Negative Log-Likelihood (NLL) as the metric to judge performance. This is because we empirically found the loss to correlate directly with sample quality. In particular, we ablate the choice of our position encodings, the number of layers and training with gradient clipping. We also include a discussion of the masking strategy and transfer learning.

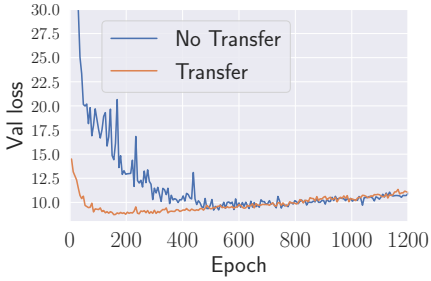


(a) Adding additional tokens helps the decoder to better align the input and the output.

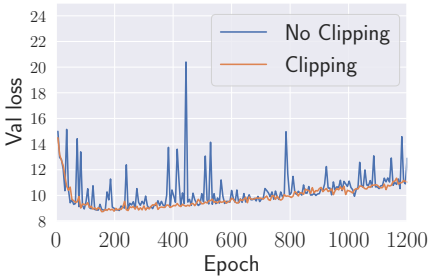


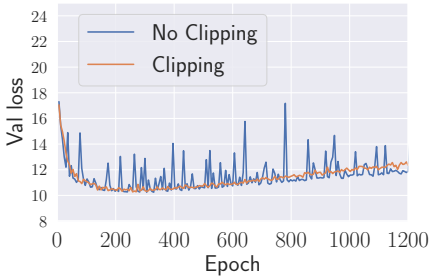
(b) The smaller models perform worse, but adding more layers does not yield large correspondingly larger gains.



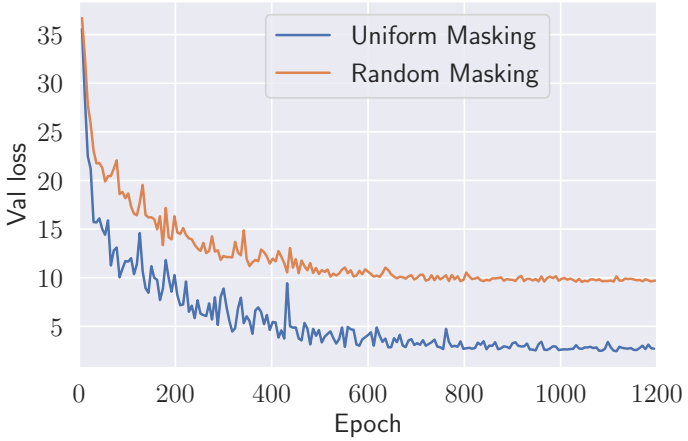


(c) **Left:** Transfering weights on the LIBRARY dataset. **Right:** Transferring weights on the LIVING dataset.





(d) **Left:** Gradient clipping applied on the DINING dataset. **Right:** Gradient clipping applied on the LIVING dataset.



(e) Using a uniform masking ratio of 0.15 shows good performance in terms of NLL, but is unable to sample owing to large distribution shift between training and inference.

Figure 7: **Ablation Studies** We show the validation losses for the different architectural choices we make.

E.1 POSITION ENCODINGS

We consider the input conditioning to be a set. In contrast, the output is a sequence. Thus, the model needs additional information to *align* the input and the output . We use *object index tokens* $\mathcal{O}^i$ and the *relative position tokens* $\mathcal{R}^i$ to provide this additional information.

During training, the objects themselves are permuted. The intuition is that $\mathcal{O}^i$ injects information about how *early* or *late* each object must appear in the output sequence. However, this information alone is not enough to disambiguate where each of the attributes of the object must appear. Hence, we also add $\mathcal{R}^i$ to the object attribute embeddings. Together, these embeddings localize the position of the attribute in the output sequence, given the current permutation.

In Fig. 7a, we progressively add our embeddings to the Baseline model which is the model without any positional encodings on the encoder. It is clear that using our embeddings helps the model better *align* the set-input and the sequence-output. While each of $\mathcal{O}^i$ and $\mathcal{R}^i$ roughly align the input and the output, it is only when using both the embeddings that the model can precisely locate the actual position of tokens in the output sequence.

E.2 NUMBER OF LAYERS

For all the experiments in the main paper, we used 4 transformer layers in both the encoder and the decoder. In Fig. 7b, we show how the model performance scales with scaling the number of layers. We see that the performance correlates strongly with the number of layers. However, the performance gains become marginal when going from 4 to 8 layers or 8 to 16 layers. These larger models take longer to train and sample from. We believe our 4 layer models provide a good compromise between performance and speed.

Note that the values in Fig. 7b are smoothed by an interpolating spline to highlight the general trend.

E.3 GRADIENT CLIPPING

We found that the validation loss oscillated considerably during training. Upon further investigation, we noticed that the gradients norms tended to be unusually large, especially for the last layers in the parameter generating MLPs. Thus, we train the final networks with gradient clipping. Surprisingly, we found that even without gradient clipping, if we retain the model with the best NLL on the validation set, the performance is the same. However, with gradient clipping, we found the training curves to be much smoother (Fig. 7d). Consequently, we were able to perform validation at less frequent intervals to select the best performing model, which sped up training.

E.4 MASKING STRATEGY

MaskGIT (Chang et al., 2022) find that using a robust masking strategy is important, as the usual 15% masking leads to a distribution shift between training and sampling. We see in Fig. 7e that masking with a uniform ratio of 15% leads to better NLL as the network is more confident in it's predictions.

But we found out that the we could not sample from such a trained network, as it would output a stop token after only generating a few objects, which intuitively makes sense, as the network would only see a few mask tokens during training.

E.5 TRANSFER LEARNING:

We plot the validation loss in Fig. 7c on the LIBRARY and LIVING datasets under two configurations - No Transfer, where the models are trained from scratch and Transfer, where the model is first trained on the BEDROOM dataset and these weights are used as initialization for training on the target dataset. We make a few observations: 1. The models begin to overfit fairly early. For the BEDROOM dataset, the loss contiues to fall until epoch 1200, but in the No Transfer configuration for the LIBRARY dataset, we see overfitting at epoch 150 and for the LIVING dataset, at epoch 600. We hypothesize that this is due to the small size of these datasets compared to the BEDROOM dataset. 2. The No Transfer configuration has a higher (worse) NLL as compared to the Transfer configuration, even when trained for longer.

These observations led us to use the Transfer configuration for the LIBRARY, LIVING and DINING datasets.

F SAMPLING DETAILS

We highlight the difference between our sampling algorithm and the standard conditional sampling algorithm in this section. These differences are highlighted in blue in Alg. 1. The primary difference is that in our sampling algorithm, a forward pass is made through the decoder every time a new token is sampled. This token then replaces the corresponding `[MASK]` token in both C and S.

In addition, our algorithm runs for a fixed number of iterations (until all `[MASK]` tokens are replaced) compared to the standard algorithm which terminates when an `EOS` token is generated. This is both an advantage and a drawback - it is an advantage in the sense that a user can implicitly specify the number of objects by specifying the number of `[MASK]` tokens. It is a drawback in that the number of objects must be known before sampling can proceed.

F.1 A SAMPLING TRICK

For our outlier detection examples, we use a simple trick - if there is only a single object to be sampled, we can create a permutation so that the `[MASK]` tokens of the object to be sampled are toward the end of the sequence in C and S. With this permutation we only have to make forward passes beginning from the first masked token. All the tokens before the first masked token can simply be copied. This leads to faster sampling.

Algorithm 1 Standard Cond. Sampling

Require: $C = (c_i)_{i=1}^{k}$, $S = (\texttt{SOS})$, $s = \phi$
$C^g = g_\phi([\mathcal{I}, C])$ ▷ Only performed once
while $s \neq \texttt{EOS}$ **do**
  $s = \mathsf{SAMPLE}(f_\theta(S_{<i}, C^g))$
  $S.\mathsf{append}(s)$ ▷ C not updated
end while
return S

Algorithm 2 Our Sampling

Require: $\mathcal{I}, C = (\texttt{[MASK]})_{i=1}^{k}$, $S = (\texttt{SOS})$
for $i \leftarrow 1$ to k **do**
  $C^g = g_\phi([\mathcal{I}, C])$
  $s = \mathsf{SAMPLE}(f_\theta(S_{<i}, C^g))$
  $C[i] = s,\ S.\mathsf{append}(s)$
end for
return S

In all our experiments, we set the number of objects to be sampled to be the same as the number of objects in the ground-truth layout associated with the particular floorplan boundary.

G GENERATION TIMES AND PARAMETER COUNTS

In Table 1, we compare the time required to generate a layout to FastSynth (Ritchie et al., 2019), SceneFormer (Wang et al., 2021), and ATISS (Paschalidou et al., 2021). Additionally, we compare the number of parameters for each method. The generation times for the other models are taken from ATISS. We train our models on an NVIDIA A100 GPU. To ensure fair comparison to the numbers in ATISS, we ran inference on a GTX 1080 GPU which is the same GPU used in ATISS.

Table 1: **Comparison of generation times and parameter counts**: We compare the time required to generate a single scene for different categories and the number of parameters required by each model. Our model requires significantly less time and fewer parameters than existing methods. Slightly over 50% of our parameters are used by the boundary encoder.

	Synthesis Time (ms) (↓)				Params ($\times 10^6$) (↓)
	BEDROOM	LIVING	DINING	LIBRARY	
FastSynth	13193.77	30578.54	26596.08	10813.87	38.1
SceneFormer	849.37	731.84	901.17	369.74	129.29
ATISS	102.38	201.59	201.84	88.24	36.05
Ours	**33.69**	**77.37**	**76.75**	**29.32**	**19.44**

Satisying constraints: ATISS cannot constrain on future tokens. However, ATISS can still be used to generate constrained layouts by performing rejection-sampling (Gentle, 2000). This means, we discard any samples that do not satisfy our given constraints. This is obviously quite inefficient. In Fig. 8, we quantify how inefficient ATISS is in comparison to COFS. The x-axis is the number of constrains we enforce, and the y-axis is the average number of samples needed to satisfy those constraints. This is performed over the whole test set of BEDROOM and the constraints are chosen to be locations of the Ground Truth objects. A sample is considered a succes if it is whithin $\epsilon = 0.01$ of the constraint. We see that for COFS, the number is almost constant regardless of the number

of constraints. One thing to note however is that this test is only for checking the satisfaction of constraints, we did not measure the quality of generated samples, and leave that for future work.

We see that COFS almost always needs a single sample to satisfy the constraints. The number is higher than 1 because of a few reasons - 1. sampling is inherently random and sometimes the samples are produced which are further than the threshold ϵ, 2. L-shaped boundaries sometimes provide strong priors which makes the model ignore the condition. ATISS, on the other struggles as the number of constraints increases. This is expected - with increasing number of constraints, the search space grows exponentially. Overall, our proposed method is many orders of magnitude faster than ATISS.

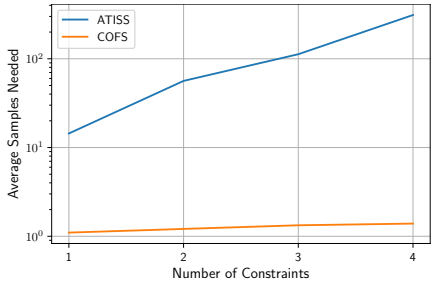


Figure 8: **Constraint Satisfaction:** The generated samples are not guaranteed to respect the provided conditioning. We see that ATISS in general need orders of magnitude more samples in order to satisfy given location constraints. Furthermore, the number of samples needed grows exponentially with the number of constraints imposed.

H Attribute-Level Conditioning

We first recap the sampling strategy of ATISS.

$$c_\theta : \mathbb{R}^{64} \rightarrow \mathbb{R}^{C} \qquad \hat{\mathbf{q}} \mapsto \hat{\mathbf{c}} \tag{5}$$

$$t_\theta : \mathbb{R}^{64} \times \mathbb{R}^{L_c} \rightarrow \mathbb{R}^{3\times 3\times K} \qquad (\hat{\mathbf{q}}, \lambda(\mathbf{c})) \mapsto \hat{\mathbf{t}} \tag{6}$$

$$r_\theta : \mathbb{R}^{64} \times \mathbb{R}^{L_c} \times \mathbb{R}^{L_t} \rightarrow \mathbb{R}^{1\times 3\times K} \qquad (\hat{\mathbf{q}}, \lambda(\mathbf{c}), \gamma(\mathbf{t})) \mapsto \hat{\mathbf{r}} \tag{7}$$

$$s_\theta : \mathbb{R}^{64} \times \mathbb{R}^{L_c} \times \mathbb{R}^{L_t} \times \mathbb{R}^{L_r} \rightarrow \mathbb{R}^{3\times 3\times K} \qquad (\hat{\mathbf{q}}, \lambda(\mathbf{c}), \gamma(\mathbf{t}), \gamma(\mathbf{r})) \mapsto \hat{\mathbf{s}} \tag{8}$$

These equations say the following: *From a query vector* $\hat{\mathbf{q}}$*, the model predicts a class. From the query and class, the model predicts the translation. From the query, class, and translation, the model predicts a rotation, and so on.* This means that in ATISS, *future* attributes cannot affect the distribution of previous attributes. When conditioning, we can specify the class and then sample a translation, but we cannot specify a translation and let the model infer the most likely class for that given translation.

In contrast, COFS has bidirectional attention on the encoder side, enabling us to specify *any* subset of object attributes. This is done by replacing the `[MASK]` token corresponding to the object attribute by its actual value in C. The copy-paste objective ensures that the same attribute will be sampled at the desired location by the decoder. The mask-predict objective trains the model to get the most-likely attributes for the unspecified tokens.

We describe the process using the following example: We start out with a layout, shown in Fig. 10a. If we mask out the table in cyan (Fig. 10b, and sample unconditionally, we get another similar table (Fig. 10c). We now wish to have some control over the generation process.

We now mask out a different object - stool in the upper left corner. We have masked out a single object, thus we have 8 `[MASK]` tokens. Our sequences C and S look like Fig. 10d. If we want to specify the position of the next object, we simply set the token corresponding to position-attribute of the next object in C - c_i to the value we want. We show a few examples of this type of conditioning in Fig. 10f and Fig. 10g. In the rest of the figures, before beginning sampling, we set the class tokens. We see that the generated layouts follow the condition, while also generating plausible layouts, even if the classes of conditioning objects never occur together. As an example, there are only 5 examples

of bedrooms with two beds, yet our model is able to reason about the placement of such challenging layouts in Row 5.

We further see that the model is able to place other objects in such a manner that the constrained objects can still satisfy their constraints. In Row 4, we see that when we constrain the angle of the bed, the other objects move in tandem to create a plausible layout.

H.1 An illustrative example

In this section, we show how our attribute-level conditioning might be used in practice. In Fig. 9 we show some edits that a user might be interested in that can be performed by our system. We start out with a GT layout and show the unconditional ATISS generated layout. ATISS does not offer control over the angles of the generated objects, as it requires the user to specify the location before generating the angle attribute.

On the other hand, COFS offers such control. In the following examples, we retain the class and sizes of the objects, and change the angles so that the layout is changed. We show that COFS can target individual attributes and generate the remaining attributes so that the generated layout is realistic.

(a) This is an unconditional sample from COFS.

(b) Conditional COFS sample where we change the orientation of the `bed`, `nightstands` and `wardrobe`.

(c) Conditional COFS sample where we change the orientation of the `bed` and `nightstands`, but retain the orientation of the `wardrobe`.

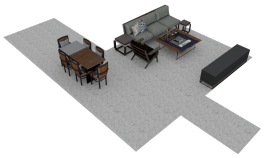
(d) The GT layout.

(e) Layout generated by ATISS, which does not provide fine-grained attribute-level control.

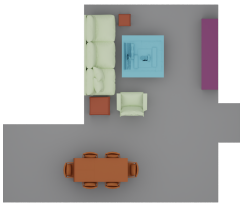
Figure 9: We show how COFS can be used to selectively edit parts of a scene. In all attribute-conditioned COFS samples, the method automatically determines the most realistic translations conditioned on *future* tokens/attributes/parameters of size and angle.

I Additional Results

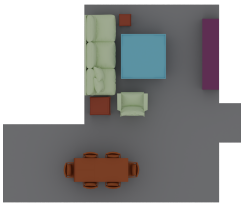
We show additional results on unconditional sampling from our model in the concluding figures. Our synthesised layouts are novel and do not merely copy the ground-truth layout. In addition, we see that our layouts respect the floorplan boundary and mimic the underlying style of the datasets, in terms of object-object co-occurrence.



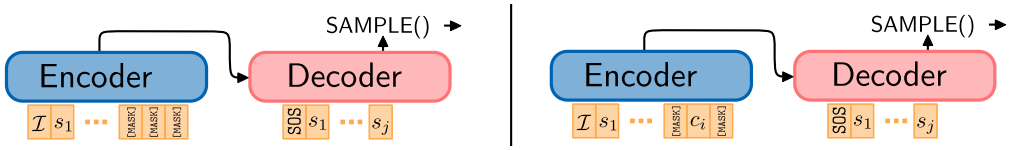
(a) Perspective view of the partial floorplan



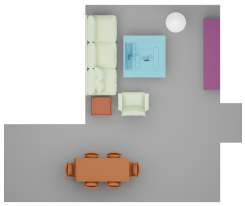
(b) Simplified Orthographic view



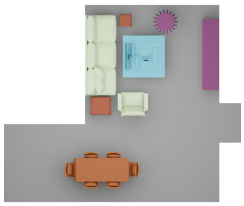
(c) Unconditionally sampled table (cyan).



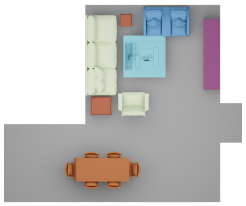
(d) **Left:** Unconditional Sampling **Right:** Conditioning by specifying attributes in C.



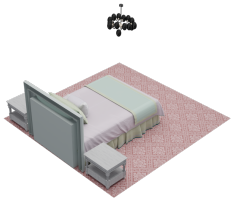
(e) We constrain the location of the object to be sampled. Shown by the white circle.



(f) Conditionally generated sample. (Purple)

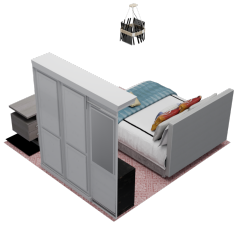


(g) We now condition the size to be *large*. The class automatically changes to satisfy the conditioning input.

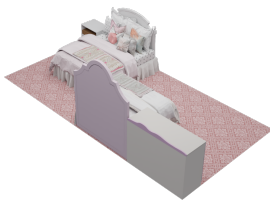


(h) Ground-truth Floorplan

(i) We fill the class tokens in C to correspond to `lamp, bed, wardrobe, table`.



(j) Same conditioning as before, but now we constrain the angle of the bed.



(k) Ground-Truth Floorplan

(l) We condition to have two beds angled opposite each other, and two nightstands.

(m) We change one nightstand to be a wardrobe.

Figure 10: Different modes of arbitrary conditioning.

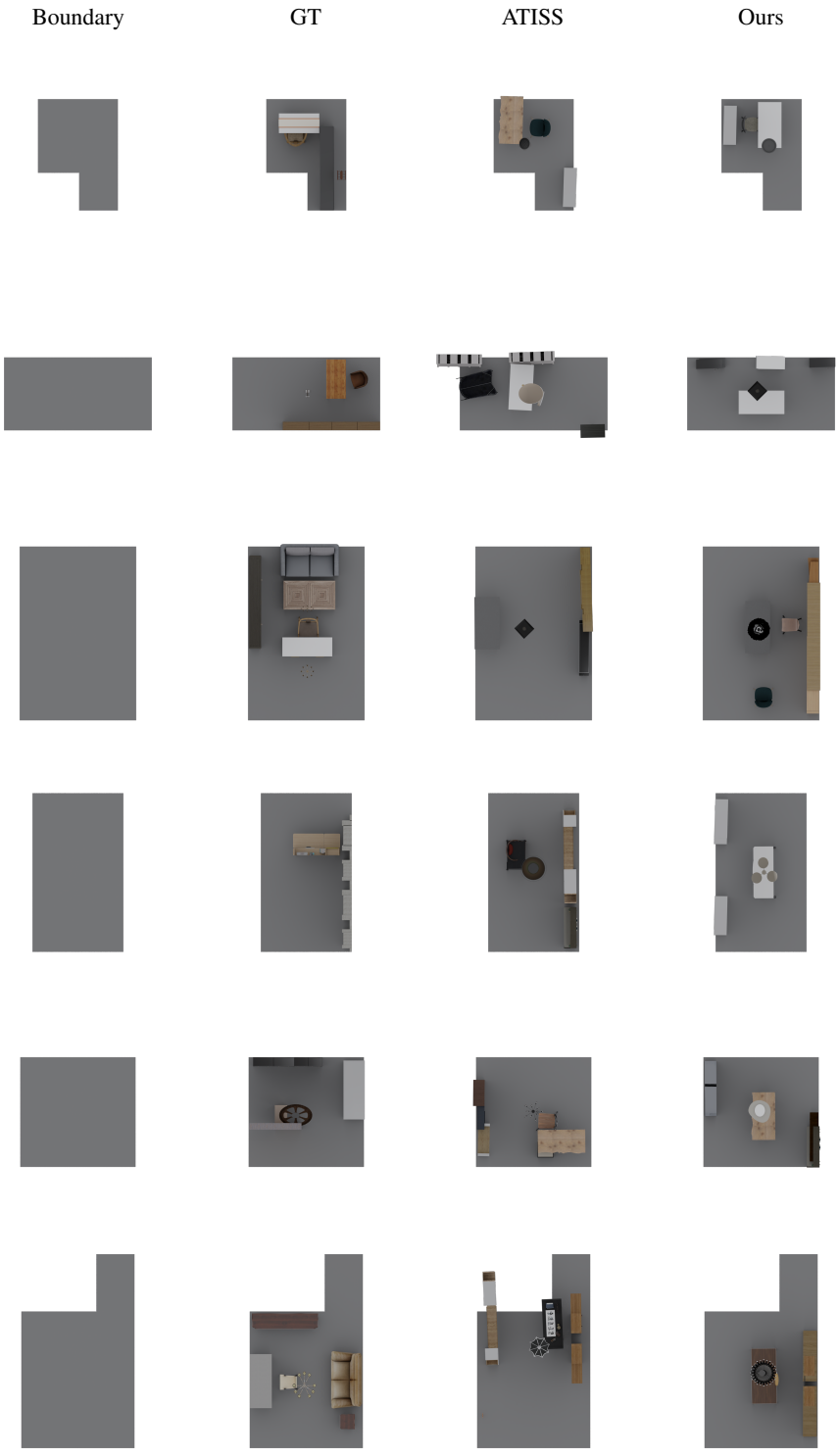


Figure 11: **Scene generation from scratch:** We compare generated scenes from GT, ATISS, and our model on LIBRARY class.

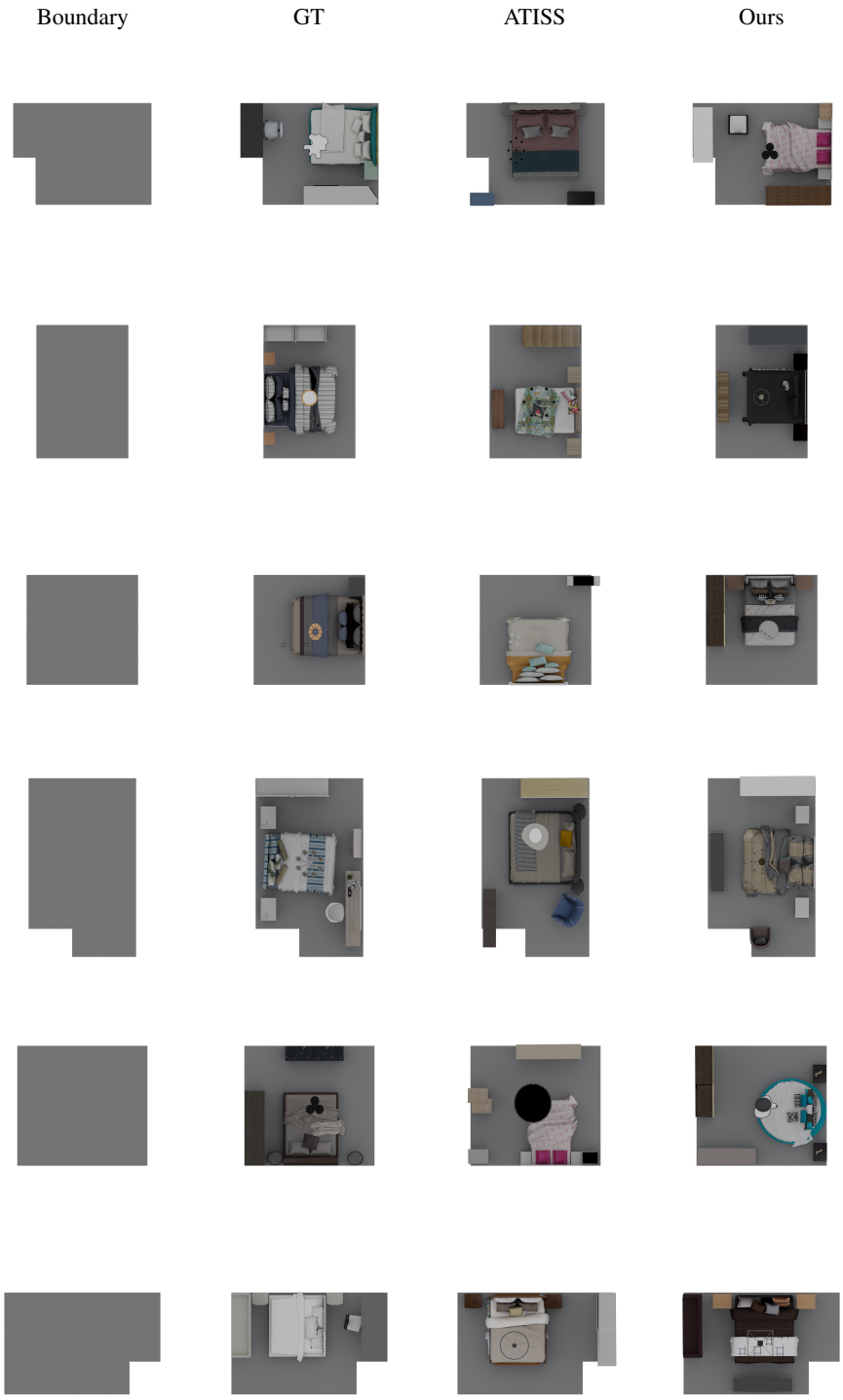


Figure 12: **Scene generation from scratch:** We compare generated scenes from GT, ATISS, and our model on BEDROOM class.

Figure 13: **Scene generation from scratch:** We compare generated scenes from GT, ATISS, and our model on DINING class.

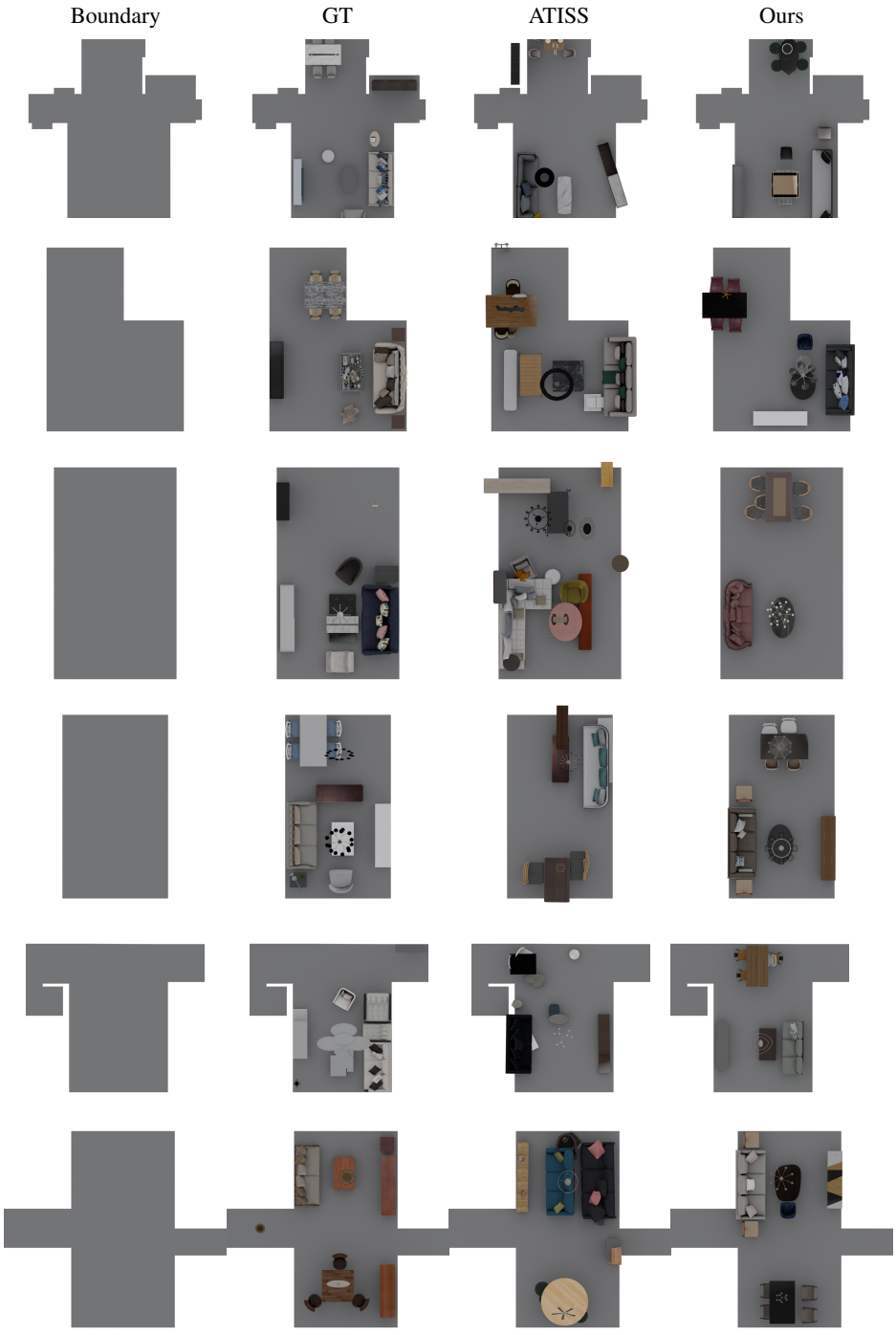


Figure 14: **Scene generation from scratch:** We compare generated scenes from GT, ATISS, and our model on LIVING class.

Table 2: Summary of key notation used in the paper.

Symbol	Description
$\mathcal{I}$	A binary image representation of the floorplan boundary.
g_ϕ	The Condition Encoder. Implemented as Transformer Encoder with Bidirectional Attention.
f_θ	The Generative Model. Implemented as Transformer Decoder with Causal Attention.
$g_\psi^{\mathcal{I}}$	The Boundary Encoder. An untrained ResNet-18 model.
$\mathcal{M}$/`[MASK]`	A learnable token representing a missing value which the Generative Model tries to predict.
C	The sequence of tokens describing the condition. It is the input to the Condition Encoder.
c_i	The i-th element of C.
C^g	The output of the last layer of the Condition Encoder. Encodes conditions from C and boundary $\mathcal{I}$.
S	The sequence representing the layout.
S^{GT}	The sequence representation of the Ground Truth layout.
s_i	The i-th element of S.
b	Generic placeholder for attribute/property of an object.
τ	Type/class attribute. A single integer
t	Translation attribute. A vector with three entries.
e	Extent/size attribute. A vector with three entries
r	Rotation attribute. A vector with a single entry. (rotation around z)
$\mathcal{O}^i$	Object Index Token. Each attribute of an objects gets the same token, which helps the network associate different tokens with different objects. The number of these tokens is the maximum number of objects in any scene in the dataset..
$\mathcal{R}^i$	Relative Position Token. Each attribute of an objects gets a different but shared token, which helps the network associate different tokens with individual object attributes. The number of these tokens is the number of attributes ($8 = 1\tau + 3t + 3e + 1r$).
$\mathcal{P}^i$	Absolute Position Token. Standard positional embedding/token. Each element in the flattened sequence representation of a scene gets a token.

REFERENCES

Huiwen Chang, Han Zhang, Lu Jiang, Ce Liu, and William T. Freeman. Maskgit: Masked generative image transformer. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2022.

Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018. URL http://www.blender.org.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Huan Fu, Bowen Cai, Lin Gao, Ling-Xiao Zhang, Jiaming Wang, Cao Li, Qixun Zeng, Chengyue Sun, Rongfei Jia, Binqiang Zhao, and Hao Zhang. 3d-front: 3d furnished rooms with layouts and semantics. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 10913–10922, 2021. doi: 10.1109/ICCV48922.2021.01075.

James Gentle. Monte carlo statistical methods by c. p. robert; g. casella. *Journal of the Royal Statistical Society. Series D (The Statistician)*, 49:632–633, 01 2000. doi: 10.2307/2681053.

Dan Hendrycks and Kevin Gimpel. Gaussian Error Linear Units (GELUs). *arXiv e-prints*, art. arXiv:1606.08415, June 2016.

A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2020.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014. URL http://arxiv.org/abs/1404.5997.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7871–7880, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.703. URL https://aclanthology.org/2020.acl-main.703.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

Gaurav Parmar, Richard Zhang, and Jun-Yan Zhu. On aliased resizing and surprising subtleties in gan evaluation. In *CVPR*, 2022.

Despoina Paschalidou, Amlan Kar, Maria Shugrina, Karsten Kreis, Andreas Geiger, and Sanja Fidler. Atiss: Autoregressive transformers for indoor scene synthesis. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Daniel Ritchie, Kai Wang, and Yu-An Lin. Fast and flexible indoor scene synthesis via deep convolutional generative models. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6175–6183, 2019. doi: 10.1109/CVPR.2019.00634.

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, pp. 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Alex Wang and Kyunghyun Cho. Bert has a mouth, and it must speak: Bert as a markov random field language model. *arXiv preprint arXiv:1902.04094*, 2019.

Xinpeng Wang, Chandan Yeshwanth, and Matthias Nießner. Sceneformer: Indoor scene generation with transformers. In *2021 International Conference on 3D Vision (3DV)*, pp. 106–115. IEEE, 2021.

Tomer Weiss, Alan Litteneker, Noah Duncan, Masaki Nakada, Chenfanfu Jiang, Lap-Fai Yu, and Demetri Terzopoulos. Fast and scalable position-based layout synthesis. *IEEE Transactions on Visualization and Computer Graphics*, 25(12):3231–3243, 2019. doi: 10.1109/TVCG.2018.2866436.