Generative Layout Modeling using Constraint Graphs Supplementary Material

Wamiq Para[1] Paul Guerrero[2] Tom Kelly[3] Leonidas Guibas[4] Peter Wonka[1]
[1]KAUST [2]Adobe Research [3] University of Leeds [4] Stanford University
{wamiq.para, peter.wonka}@kaust.edu.sa guerrero@adobe.com twakelly@gmail.com guibas@cs.stanford.edu

Abstract

In this document, we collect additional results and provide more details on several aspects of the method, including furniture layout generation and the network architecture. In Section 1, we describe furniture layout generation in more detail and show additional furniture layout generation results in Section 2. Then, we show experiments with generating both rooms and furniture of a floor plan in Section 3. Additional results conditioned on the count or existence of rooms are presented in Section 4. We give more details on our data pre-processing in Section 5, show the interface of our perceptual study in Section 6, and describe additional details of our architecture in Section 7.

1. Furniture Layout Implementation Details

In this section, we describe the implementation details for furniture layouts that differ from floor plans. Since furniture layouts are less constrained than floor plans (furniture pieces do not need to cover all of the layout without gaps, for example), we do not add constraining edges and omit the optimization step, directly using the element constraints as elements instead: $N = N^C$. We train one furniture layout model that we condition on the room type, and optionally on the room width, height and door locations.

Layout representation Each element represents a piece of furniture with an oriented bounding box $N = (\tau, x, y, w, h, \alpha)$, with furniture type τ, bounding box position (x, y), width and height (w, h), and orientation α.

Element constraints The element constraint model described in Section 3.2 of the main paper generates constraints $N^C = (\tau, x, y, w, h, \alpha)$ for all parameters of a furniture piece that are directly used as furniture pieces N. The orientation α has a different value range than the other parameters, and we quantize it to 5 bits instead of the 6 bits we use for the other parameters.

Table 1. Free generation of furniture layouts. We compare the layout statistics of our results to a simple procedural model and to purely image-based generation with StyleGAN [3]. Our method shows a clear improvement over both baselines.

method	$\hat{s}_t$	$\hat{s}_r$	$\hat{s}_a$	$\hat{\mathbf{s}}_{\mathbf{avg}}$
StyleGAN	16.50	6.24	7.09	9.94
Procedural	15.65	5.21	4.90	8.59
ours free	**1.00**	**1.00**	**1.00**	**1.00**

2. Additional Furniture Layout Results

In this section, we present additional furniture layout results. We generated approximately 10k furniture layouts for all room types in our floor plans. We evaluate these furniture layouts using the layout statistics described in Section 4 of the main paper. To compute topological statistics $\hat{s}_t$, we create an r-NN graph of the furniture pieces as layout graph, with $r = 15\%$ of the layout diagonal. Thus, topological statistics capture relationships in local neighborhoods of furniture pieces, for example which types of furniture are typically placed next to each other.

Metrics Since elements in furniture layouts have additional parameters, we extend the list of layout statistics. We add one statistic to the shape statistics S_r:

s_r^o: a histogram of orientation distributions for each element type.

And the alignment statistics S_a are extended with:

s_s^o: a histogram of the differences between orientations.
s_s^w: a histogram of the differences between widths.
s_s^h: a histogram of the differences between heights.

Baselines We compare the furniture layouts to two baselines: layouts generated with StyleGAN and layouts generated with a simple procedural model that is fitted to dataset statistics. For *StyleGAN* we proceed similar as in the floor plan setting: we render our furniture layout dataset, train

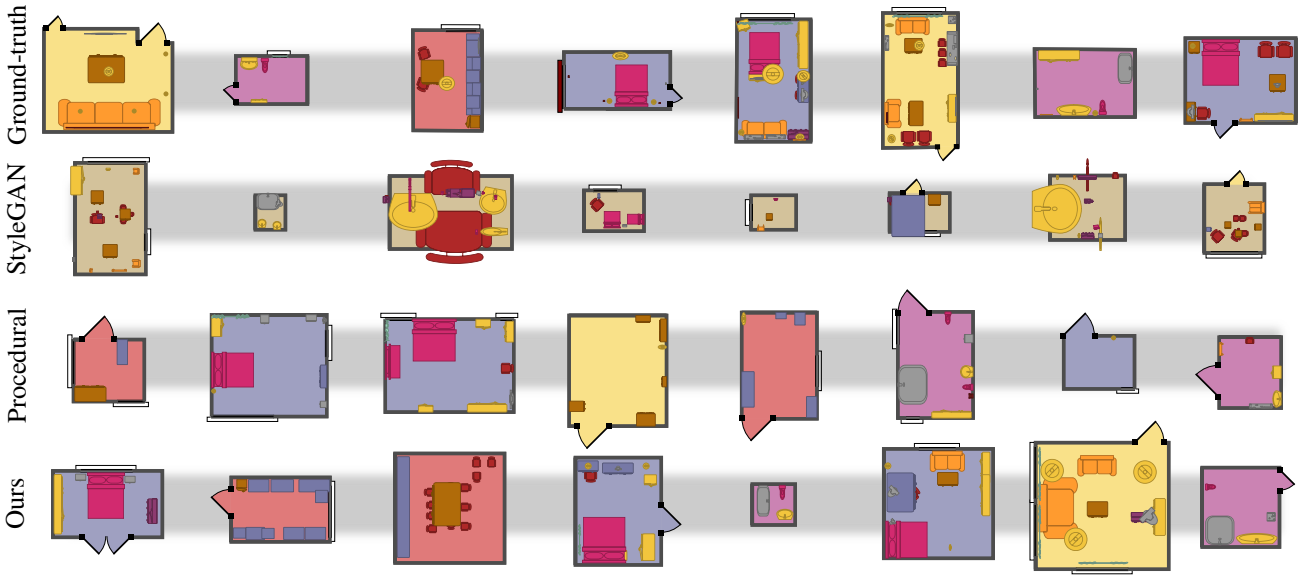

Figure 1. Furniture layout results for the ground truth, StyleGAN, the simple procedural model and our approach. Note that StyleGAN is not conditioned on the room type in these experiments and thus freely generates different room types, according to its learned distribution.

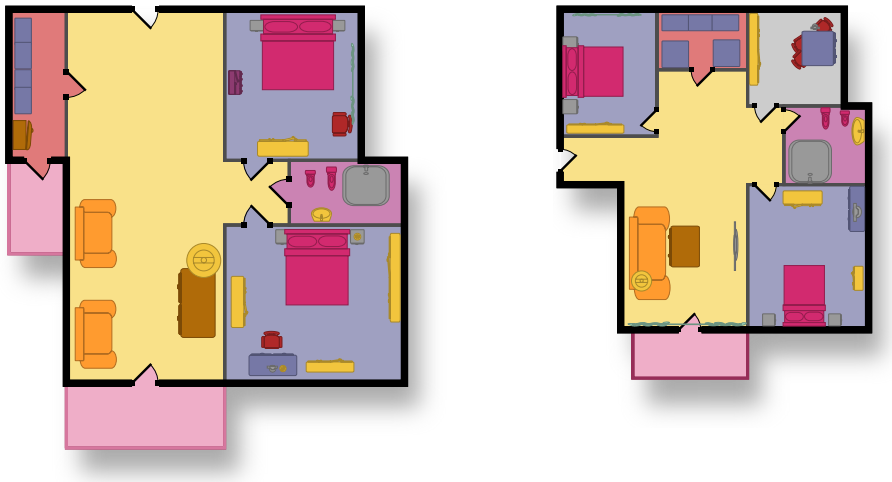

Figure 2. Furniture generation conditioned on room shape and doors allows for the generation of furnished rooms. Room floors and furniture are colored according to the their type.

StyleGAN, and parse the generated images back into furniture layouts. With the procedural model we want to show that a hand-crafted procedural model that can reproduce a given dataset distribution is not trivial to design, even when fitting the model to dataset statistics. The *procedural* model draws the room size from the dataset distribution of room sizes, and proceeds similarly for the number of doors, windows and number of objects from each furniture category. These distributions are independent and conditioned only on room type. Furniture width and height is sampled from a distribution conditioned on the furniture category. Doors, windows, and furniture are positioned around the perimeter of room at a random location, re-trying up to 10 times in the event of a collision.

Table 1 and Figure 1 show the results of this comparison. Similar to floor plans, our method shows a clear advantage over the purely image-based StyleGAN and over the simple procedural model.

3. Furnished Floor Plans

Similar to floor plans, we can condition furniture layouts on constraints like the room shape, and the door position. This allows us to place furniture layouts into rooms of pre-

Figure 3. Additional generated floor plans conditioned on room count or existence. We show four examples for each condition. Statistics over room count/existence for these conditioned results compared to unconditioned results and the dataset are shown on the right. While our method does not fully guarantee satisfying the condition, it does satisfy the condition with high probability.

viously generated floor plans, resulting in fully generated furnished floor plans. Two furnished floor plans are shown in Figure 2. Note that the furniture layouts correctly respect the shape and door location of the room they were placed in.

4. Additional Floor Plans Conditioned on Room Count or Existence

We show additional qualitative results of generated floor plans that were condition on given room counts or room existence in Figure 3. On the left and center we show four additional floor plan samples for each condition that was shown in Figure 7 of the main paper. On the right, we provide quantitative results that compare the probability distribution of different room counts (top row), or room existence (bottom row) for floor plans from the testset, unconditionally generated floor plans, and floor plans generated with each of the two corresponding conditions. Since the condition is not a hard constraint, it is not guaranteed to be satisfied by our method, but our approach is successfully trained to satisfy it with high probability.

5. Data Preparation

Like other methods, we first re-scale and quantize all floorplans to a uniform coordinate grid, 64x64 in our experiments, corresponding to 6-bit quantization. To decompose a floor plan into rectangular boxes, we first construct a grid by extending all vertical and horizontal edges into infinite lines. This grid gives us an over-segmentation of the floor plan into boxes. To reduce the number of boxes, we perform several rounds of merging adjacent boxes if they have the same room type and if the merged result is still a rectangular box. We perform one round of merging vertically adjacent boxes, followed by three rounds of merging horizontally adjacent boxes. After merging, we obtain our final boxes. We can then create all constraint edges and descriptive edges based on the adjacency and door connectivity of these boxes.

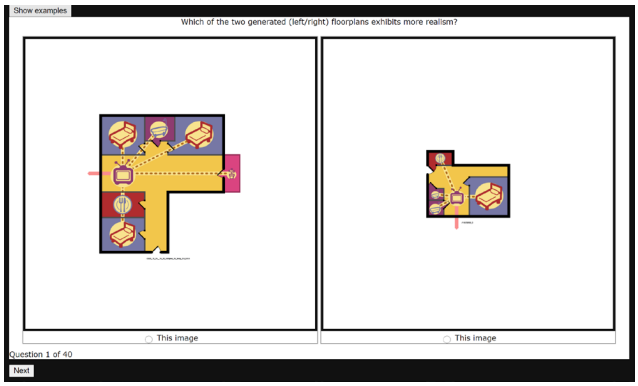

Figure 4. A screen shot of our perceptual study. Participants were asked to compare the realism of the two floor plans shown on the left and right.

6. Perceptual Study Screenshot

A screenshot of the perceptual study is shown in Figure 4. We also provided a legend of room colors and icons and a few examples of ground truth floor plans under the button labeled 'Show examples'.

7. Architecture Details

In this section, we describe the architecture of our generative models in more detail.

The model for element constraint generation consists of 12 Transformers blocks. Our sequence lengths depend on the particular dataset used, and are listed further below. The edge generation model is a Pointer Network with two-parts: 1. An encoder which generates embeddings, and can attend to all elements in the sequence of element constraints and 2. A decoder which generates pointers, and can attend to elements in an autoregressive fashion. In our experiments, we use an encoder with 16 layers and a decoder with 12 layers. We use 384 dimensional embeddings in all our models.

Constrained generation is performed by a variant of the

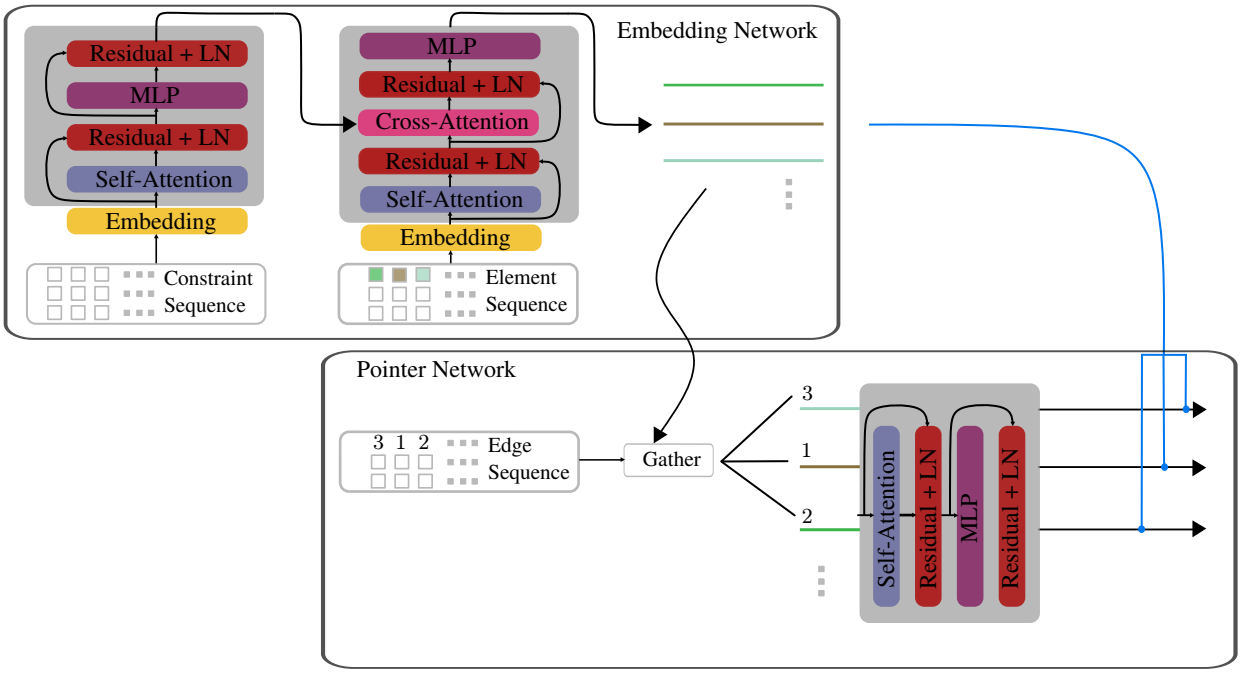

Figure 5. The user-constrained Edge Generation Model. An embedding function modeled by a transformer (top left) generates element embeddings that are re-arranged based on the edge sequence. This sequence is ingested by the edge model (bottom right), which is also implemented as a transformer. The encoder (left block in the embedding network) is only used when performing constrained generation.

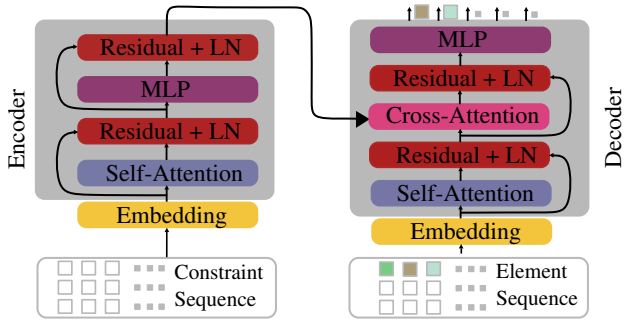

Figure 6. The constrained element generation model. Having an unmasked encoder allows our network to attend to all elements of the constraint sequence. The sequences have three parts - the value sequence, the position sequence and the type sequence (see Section 7.1).

unconstrained models. Concretely, we add a constraint encoder to both the element constraint model and the edge models resulting in an encoder-decoder architecture, see Figure 6 for an illustration. In the edge models, we concretely change the encoder of the Pointer Network to an encoder-decoder architecture, as illustrated in Figure 5. The constraint encoder is a stack of Transformer blocks allowed to attend all elements of the constraint sequence. The decoder is another stack of blocks allowed to attend to all tokens in the constraint sequence. We use 8 layers for constraint encoder in the element model and 3 layers in the edge-model.

Both the element constraint model and the edge model are composed of embedding blocks (yellow in Figures 6 and 5) and transformer blocks (gray). Next, we describe these building blocks in more detail.

7.1. Embedding

Element Constraint Model The input sequence to the element constraint model has three components - the value sequence $S_E = \{v_i\}_{i=1}^{kM_N}$, the position sequence $I = \{i\}_{i=1}^{kM_N}$ and the type sequence $T = \{i \bmod k\}_{i=1}^{kM_N}$, where M_N is the number of elements and k is the number of properties per element. We use three separate learned embeddings, one for each sequence. The final embedding is the sum of these three embeddings.

Edge Model The edge model operates on sequences of learned element embeddings g_{θ_ρ}, as described in Section 3.3 of the paper. The embedding function is modeled by a transformer with the same architecture as the element constraint model, that takes as input the element constraint sequence and outputs a sequence of element embeddings. Similar to the element constraint model, the embedding function can be conditioned on a sequence of constraints by adding an encoder, as shown in the top left of Figure 5. The sequence of element embeddings is then arranged according to the

edge sequence (concatenating the element embeddings corresponding to the two elements of each edge) and processed by the edge model (Figure 5, right) as described in Section 3.3 of the paper.

Bedroom/Balcony Conditioning This is one of the the examples of conditional generation shown in the main paper. In both of these cases the constraint sequence is of length 3, consisting of a start token, followed by a constraint token and ending with a stop token. For controlling the balcony generation, the constraint token is either 1 or 0 and for controlling the number of bedrooms, the condition token contains the number of bedrooms. At training time, we obtain the values for these tokens from the ground truth floor plan, at inference time, the tokens can be set by the user.

7.2. Transformer GPT2 Blocks

We use Dropout [5] with a drop probability of 0.2 immediately after performing the sum of embeddings. The attention layers in all our experiments use Multiheaded Attention with 12 heads. We set our embedding dimension $d = 384$.

Encoder We use a stack of standard GPT-2 [4] encoder blocks. The MLP block inside the encoder (and the decoder) performs the following operation on an input tensor x

$$x = \text{Linear}(\text{GELU}(\text{Linear}(x))) \tag{1}$$

The activation function we use between the linear layers is the GELU [2] function. The first linear layer changes the embedding dimensions internally from d to $4d$. The second then goes back from $4d$ to d.

Decoder The activation $h^{(L)}$ obtained at the last layer of the encoder is used for performing cross-attention in the Decoder. We can write the operations of a Decoder block as:

$$n^{(i)} = \text{LN}(h_D^{(i)}) \tag{2}$$

$$a^{(i)} = \text{LN}\left(n^{(i)} + \text{SelfAttention}(n^{(i)}, n^{(i)})\right) \tag{3}$$

$$b^{(i)} = \text{LN}\left(a^{(i)} + \text{CrossAttention}(a^{(i)}, h^{(L)})\right) \tag{4}$$

$$h_D{}^{(i+1)} = b^{(i)} + \text{MLP}(b^{(i)}), \tag{5}$$

where LN denotes Layer Normalization [1]. We a add a single linear layer after both the Encoder and the Decoder to produce logits. The encoders are only used for constrained generation, such as floor plan or furniture generation constrained on a given floor plan boundary. In free generation, we do not have any constraints, so we do not add encoders to any of the models.

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016. 5

[2] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016. 5

[3] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *CoRR*, 2018. 1

[4] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019. 5

[5] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 2014. 5