Generative Layout Modeling using Constraint Graphs

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

We propose a new generative model for layout generation. We generate layouts in three steps. First, we generate the layout elements as nodes in a layout graph. Second, we compute constraints between layout elements as edges in the layout graph. Third, we solve for the final layout using constrained optimization. For the first two steps, we build on recent transformer architectures. The layout optimization implements the constraints efficiently. We show three practical contributions compared to the state of the art: our work requires no user input, produces higher quality layouts, and enables many novel capabilities for conditional layout generation.

1. Introduction

We study the problem of topologically and spatially consistent layout generation. This problem arises in image layout synthesis, floor plan synthesis, furniture layout generation, street layout planning, and part-based object creation, to name a few. Generated content must meet stringent criteria both globally, in terms of its overall *topological* structure, as well as locally, in terms of its *spatial* detail. While our work applies to layouts in general, we focus our discussion on two types of layouts: floorplans and furniture layouts.

When assessing layouts, we must consider the global structure which is largely topological in nature, such as connectivity between individual elements or inter-element hop distance. We are also concerned with spatial detail, such as the geometric realization of the elements and their relative positioning, both local and non-local. Realism of such generated content is often assessed by comparing distributions of their properties, both topological and spatial, against those from real-world statistics.

Techniques to generate realistic content have made rapid progress in recent years due to the emergence of generative adversarial networks (GANs) [13, 71, 24, 57], variational autoencoders (VAEs) [27, 53], flow models [48, 63, 53], and autoregressive models [8]. However, satisfying *both* topological and spatial properties still remains an open challenge.

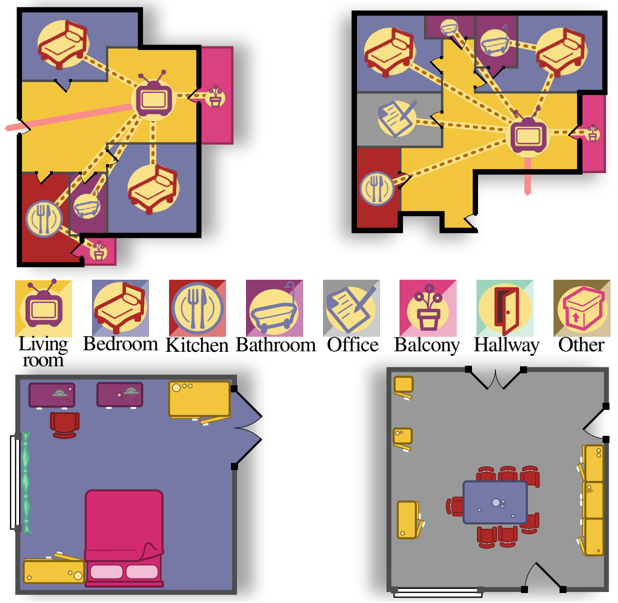


Figure 1. We present a method for layout generation. Our approach can generate multiple types of layouts, such as the floor plans in the top row, where rooms are colored by type, and furniture layouts in the bottom row, where furniture pieces are colored by type. Layouts are represented as graphs, where nodes correspond to layout elements and edges to relationships between elements. In the top row, nodes represent rooms (illustrated with room-type icons), and edges relate rooms connected by doors (dotted lines). Unlike previous methods, our method does not require any input guidance and generates higher-quality layouts.

Recently, three papers targeting this challenging problem in the floor plan setting were published [61, 16, 38]. While these papers often produce good looking floor plans, they require several simplifications to tackle this difficult problem: 1) RPLAN [61] and Graph2Plan [16] require the outline of the floorplan to be given. 2) HouseGAN [38] does not generate the connectivity between rooms that would be given by doors, and RPLAN places doors using a manually defined heuristic that is not learned from data. 3) HouseGAN and Graph2Plan require the number of rooms, the room types and their topology to be given as input in the form of an adjacency graph. 4) All three methods require a heuristic post-process that is essential to make the floorplan look more realistic, but that is not learned from data, such as adding doors and windows (RPLAN), or fixing gaps and overlaps (HouseGAN). In addition, there is still a lot of room to improve the quality and realism of the results.

In this paper, we would like to explore two ideas to improve upon this exciting initial work. First, after extensive experiments with many variations of graph-based GANs and VAEs, we found that these architectures are not well suited to tackle the problem. It is our conjecture that these methods struggle with the discrete nature of graphs and layouts. We therefore propose an auto-regressive model using attention and self-attention layers. Such an architecture inherently handles discrete data and gives superior performance to current state of the art models. While transformer-based auto-regressive models [55] just started to compete with GANs built on CNNs in image generation [41, 7] on the ImageNet [10] dataset, we will show that the gap between these two competing approaches for *layout generation* is significant.

Second, we explore the idea of generative modeling using constraint generation. We propose to model layouts with autoregressive models that generate constraint graphs: individual shapes are nodes and edges between nodes specify constraints. Our auto-regressive model first generates initial nodes, that are subsequently optimized to satisfy constraint edges generated by a second auto-regressive model. These models can be conditioned on additional constraints provided by the user. This enables various forms of conditional generation and user interaction, from satisfying constraints provided by the user, to a fully generative model that generates constraints from scratch without user interaction. For example, a user can optionally specify a floorplan boundary, or a set of rooms.

In summary, we introduce two main contributions: 1) A transformer-based architecture for generative modeling of layouts that produces higher quality layouts than previous work. 2) The idea of a generative model that generates constraint graphs and solves for the spatial shape attributes via optimization, rather than outputting shapes directly.

We demonstrate our approach in the context of floor plan generation by creating room layouts and furniture layouts for apartments (see Figure 1). Our evaluation shows that our generative model allows layout creation that matches both global and local statistics of real-world data much better than competing work.

2. Related Work

We will discuss image-based generative models, graph-based generative models, and finally models specialized to layout generation.

2.1. Image-based Generation

A straight-forward approach to generate a layout is to represent it as an image and use traditional generative models for image synthesis. The most promising approach are generative adversarial networks (GANs) [13, 22, 70, 4, 24, 25, 23]. Image-to-image translation GANs [18, 71, 72, 17, 73, 49] could also be useful for layout generation, e.g., as demonstrated in this project [5]. Alternatively, modern varitional autoencoder, such as NVAE [53] or VQ-VAE2 [47] are also viable options. Autoregressive models, e.g. [8], also showed impressive results on large-scale datasets. When experimenting with image-based GANs, we noticed that they fail to respect the relationships between elements and that they cannot preserve certain shapes (e.g. axis-aligned polygons, sharp corners).

2.2. Graph-based Generation

In order to capture relationships between elements, various graph-based generative models have been proposed [57, 28, 51, 67, 31, 29, 40]. However, purely graph-based approaches only generate the graph topology, but are missing the spatial embedding. The specialized layout generation algorithms described next often try to combine graph-based and spatial approaches.

2.3. Specialized Layout Generation

Before the rise of deep learning, specialized layout generation approaches have been investigated in numerous domains, including street networks [64, 43], parcels [2, 54], floor plans [60], game levels [66], furniture placements [68], furniture and object arrangements [12], and shelves [32]. Different approaches have been proposed for layout generation, such as rule-based modeling [45, 36], stochastic search [35, 69, 66], or integer programming [44, 43, 60], or graphical models [34, 11, 6, 21, 12, 65].

In recent years, most of the focus has shifted to applying deep learning to layout generation. A popular and effective technique places elements one-by-one, [59, 20, 9], while a different approach first generates a layout graph and then instantiates elements according to the graph [19, 58, 1]. Both of these approaches are problematic in layouts such as floor plans, that have many constraints between elements, such as zero-gap adjacency and door connectivity. In such a settings it is non-trivial to a) train a network to generate constraints that admit a solution, and b) find elements that satisfy the constraints in a single forward pass. Recently proposed methods [61, 16, 38] circumvent these problems by requiring manual guidance as input, or by requiring manual post-processing. Due to these requirements, these methods are not fully generative. Recently, Nash et al. introduced PolyGen [37], a method to generate graphs of vertices that form meshes with impressive detail and accuracy. We base our method on a similar architecture, but generate layout

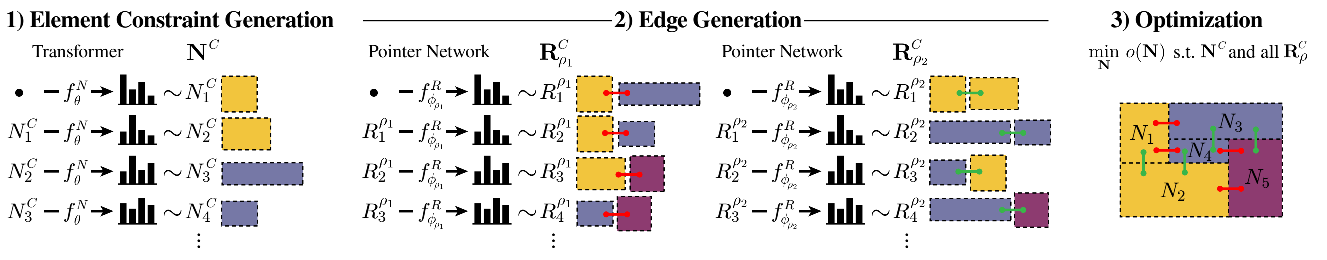


Figure 2. Overview of our layout generation approach. We generate constraints on the parameters of layout elements with a Transformer [55], and constraints on multiple types of relationships between elements using Pointer Networks [56]. Both element and relationship constraints are used in an optimization to create the final layout.

constraints instead of directly generating the final layout. Layout elements are then found in an optimization step based on the generated constraints. This gives us layouts where elements accurately satisfy the constraints.

3. Method

We present a generative model for layouts that can optionally be conditioned on constraints given by the user. Figure 2 illustrates our approach. Layouts are represented as graphs, where nodes correspond to discrete elements of the layout, and edges represent relationships between the elements. We distinguish two types of edges: *Constraining edges* describe desirable relationships between element parameters, such as an adjacency between a bedroom and a bathroom in a floor plan, and can be used to constrain these parameters. *Descriptive edges* represent additional properties of the layout that are not given by the elements, but can be useful for downstream tasks, such as the presence of a door between two rooms of a floor plan where the elements consist of rooms. Section 3.1 describes this layout representation.

A generative model can be trained to generate both layout elements and edges. However, generated elements and generated constraining edges are not guaranteed to match. For example, two elements that are connected by an adjacency edge can often be separated by a gap, or can have overlaps. As the number of constraining edges increases, the problem of generating a compatible set of edges and elements becomes increasingly difficult to solve in a forward pass of the generative model. This has been a major limitation in previous work.

We introduce two contributions over previous layout generation methods. First, we show that a two-step autoregressive approach inspired by PolyGen [37] that first generates elements and then edges is particularly suitable for layout generation and performs significantly better than current methods. We describe this approach in Sections 3.2 and 3.3.

Second, we treat element parameters and constraining edges that were generated in the first two steps as *constraints* and optimize element parameters to satisfy the generated constraints in a subsequent optimization step. In floor plans, for example, we generate constraints on the maximum and minimum widths and heights of room areas and on their adjacency, and then solve for their locations, widths and heights in the optimization step. This minimizes any discrepancies between constraining edges and element parameters. We describe the optimization in Section 3.4. In Section 3.5, we describe how to condition on user-provided constraints.

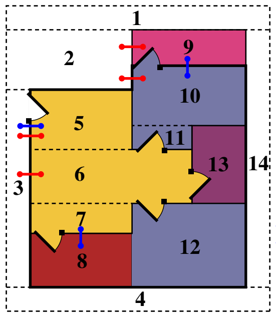
3.1. Layout Representation

We represent layouts as a graph $\mathcal{L} = (\mathbf{N}, \mathbf{R})$, where nodes correspond to layout elements $\mathbf{N}$ and edges to their relationships $\mathbf{R}$. Each layout element $N \in \mathbf{N}$ has a fixed set of domain-specific parameters. Relationship edges $R \in \mathbf{R}$ are chosen from a fixed set of edge types ρ and describe the presence of that edge between two elements $R = (N_i, N_j, \rho)$. Edges come in two groups, based on their types: constraining edges $\mathbf{R}^C$ that provide constraints for the optimization step, and descriptive edges $\mathbf{R}^D$ that provide additional information about the layout. We consider two main layout domains in our experiments: *floor plans* and *furniture layouts*, but will only focus on floor plans here. Furniture layouts are described in the supplementary material.

In floor plans, each layout element is a rectangular region of a room $N = (\tau, x, y, w, h)$, parameterized by the type of room τ, the lower-left corner of the rectangular region (x, y), and the width and height (w, h) of the region. Two types of edges in $\mathbf{R}^C$ define horizontal and vertical adjacency constraints between elements, while two types of edges in $\mathbf{R}^D$, define the presence of a wall between two adjacent elements, and the presence of a door between two adjacent elements. Multiple elements of the same type that are adjacent and not separated by a wall form a room. The set of all elements fully cover the floor plan. An example is shown in Figure 3, left. More details on both representations, including a full list of all element types, are given in the supplementary material.

3.2. Element Constraint Model

An element constraint N^C is defined as a tuple of target values for one or more of the parameters of element N. In the optimization, we will use these values as soft constraints for the corresponding parameters. We create one set of constraints for each element N of the layout. In floor plans,



Element constraint sequence

value v_i	τ_1	w_1	h_1	τ_2	w_2	h_2	...	...	s
index i	1	2	3	4	5	6	...	...	M_N
type t_i	1	2	3	1	2	3	...	...	0

Horiz. adjacency edge sequence

elem. emb. n_{ji}	n_2	n_9	n_{10}	e	n_3	n_5	n_6	...	s
index i	1	2	3	4	5	6	7	...	M_ρ
type t_i	1	2	2	0	1	2	2	...	0

Door edge sequence

elem. emb. n_{ji}	n_3	n_5	n_7	n_8	n_9	n_{10}	...	s
index i	1	2	3	4	5	6	...	M_ρ
type t_i	1	2	1	2	1	2	...	0

Figure 3. Example floor plan layout and its sequence encoding. Rooms are represented by rectangles, which are numbered and colored by room type for illustration (white for the exterior). Edges either constrain rectangles, like the red adjacency edges, or add information to the layout, like the blue door edges. Both are encoded into sequences that can be ingested by our autoregressive sequence-to-sequence models.

for example, we create constraints $N^C = (\tau, w, h)$ for the type, width, and height of each element. All continuous values are treated as range constraints, i.e. the actual values may be within the range $\pm\epsilon v^C$ of the constraint value v^C (we set $\epsilon = 0.1$ in our experiments). We use a transformer-based [55] autoregressive sequence-to-sequence model to generate these element constraints.

Sequence encoding The goal of our element constraint model is to learn a distribution over constraint sequences. To flatten our list of element constraints into a sequence of tokens, we order them from left to right first (small to large x) and top to bottom (small to large y) for elements with the same x coordinate. The ordered constraint tuples are concatenated to get a sequence of constraint values $S_E = (v_i)_{i=1}^{kM_N}$, where M_N is the number of elements in the layout and k the number of properties per element. Following PolyGen [37] we use two additional inputs per token in the sequence: the sequence index i and the type t_i of each value. Type t_i is the index of a constraint value inside its constraint tuple and indicates the type of the value (x-location, height, angle, etc.). Finally, we add a special stopping token s as last element of the sequence to indicate the end of the sequence.

Autoregressive Model Our element constraint model f_θ^N models the probability of a sequence by its factorization into step-wise conditional probabilities:

$$p(S_N; \theta) = \prod_{i=1}^{kM_N} p(v_i | v_{<i}; \theta), \tag{1}$$

where θ are the parameters of the model. Given a partial sequence $v_{<i}$, the model predicts a distribution over values for the next token in the sequence $p(v_i|v_{<i}; \theta) = f_\theta^N(v_{<i}, (1 \dots i-1), t_{<i})$, that we can sample to obtain v_i.

We implement f_θ with a small version of GPT-2 [46] that has roughly 10 million parameters. For architecture details, please refer to Section 3.6 and the supplementary material.

Coordinate Quantization We apply 6-bit quantization for all coordinate values and learn a categorical distribution over the discrete constraint values in each step of the model. Nash et al. [37] have shown that this improves model performance, since it facilitates learning distributions with complex shapes over the constraint values.

3.3. Edge Model

We generate relationship edges R between elements. These edges can be split into two sets: constraining edges $\mathbf{R}^C$ constrain element parameters during the optimization step, while descriptive edges $\mathbf{R}^D$ add information to the layout that may be needed in down-stream tasks. In floor plans, for example, adjacency edges constrain the optimization, while door and wall edges are needed to define walls and doors. We use an autoregressive sequence-to-sequence architecture based on PointerNetworks [56] to generate edges. We train one model for each of the edge types described in Section 3.1, each models the distribution for one type of edge. All models have the same architecture, but do not share weights.

Sequence Encoding To flatten the list of edges $R = (N_i, N_j, \rho)$ of any given type ρ, we first sort them by the index of the first element i, then by the index of the second element j. We then concatenate the constraints N_i^C, N_j^C corresponding to the elements N_i, N_j in each edge to get a sequence of element constraints. We use a learned embedding $n_j^\rho = g_{\phi_\rho}(N_j^C)$, giving us a sequence of element embeddings $S_\rho = (n_{j_i}^\rho)_{i=1}^{2M_\rho}$, where M_ρ is the number of edges of a given type ρ. Two additional inputs are added for each token: the index i and the type t_i, indicating if a token corresponds to the source or target element of the edge. The last token in the sequence is the stopping token s.

Due to our ordering, groups of edges that share the same source element N_i, are adjacent in the list. For types of edges where these groups are large, that is, where many edges share the same source element, we can shorten the sequence by including the constraint of a source element only once at the start of the group, and then listing only the constraints of the target elements N_j that are connected to this source element. The end of a group is indicated by a special token e. We use this shortened sequence style for the adjacency edges of floor plans.

Autoregressive Model Similar to the element constraint model, the probability of an edge sequence S_ρ is modeled by a factorization into step-wise conditional probabilities.

Unlike the element constraint model, however, the edge model $f^R_{\phi_\rho}$ outputs a pointer embedding [56]:

$$q_i^\rho = f^R_{\phi_\rho}(n^\rho_{j_{<i}}, (1 \dots i-1), t_{<i}). \tag{2}$$

We compare this pointer embedding to all element embeddings using a dot-product to get a probability distribution over elements:

$$p(n^\rho_{j_i} = n^\rho_k | n^\rho_{j_{<i}}; \phi_\rho) = \mathrm{softmax}_k \left((q_i^\rho)^T n_k \right) \tag{3}$$

that we can sample to get the index of the next element constraint in the sequence.

3.4. Optimizing Layouts

We formulate a Linear Programming problem [3] that regularizes the layout while satisfying all generated constraints:

$$\begin{aligned} \min_{\mathbf{N}} \quad & o(\mathbf{N}) \\ \text{s.t.} \quad & \mathbf{N}^C \text{ are satisfied and} \\ & \mathbf{R}^C \text{ are satisfied,} \end{aligned} \tag{4}$$

where $o(\mathbf{N})$ is a regularization term. In floor plans, for example, we minimize the perimeter of the floor plan $o(\mathbf{N}) = W + H$, where W and H are the width and height of the floor plan's bounding box. This effectively minimizes the size of the layout, while keeping the optimization problem linear. This regularization encourages compactness and a bounded layout size, favoring layouts without unnecessary gaps and holes. The definition of the constraints depend on the type of layout.

In floor plans, the x, y, w, h parameters of each element are bounded between their maximum and minimum values; we use $[0, 64]$ as bounds in our experiments. Each element constraint N^C adds constraints of the form $v^C(1-\epsilon) \leq v \leq v^C(1+\epsilon)$, for each value v^C in the element constraint N^C and corresponding value v in the element N. In our experiments, we set $\epsilon = 0.1$. Horizontal adjacency edges $R = (N_i, N_j, \rho)$ add constraints of the form $x_i + w_i = x_j$, and analogously for vertical adjacency edges.

The layout width W is computed by first topologically sorting the elements in the subgraphs formed by horizontal adjacency edges, and then defining $W := x_m + w_m$ for the last (right-most) element N_m in the topological sort. H is computed analogously. Note that we do not define $W := \max_i x_i + w_i$ to avoid the additional constraints needed to optimize over the maximum of a set. A detailed list of constraints for furniture layouts is given in the supplementary material. The challenge of designing the optimization is to keep the optimization fast and simple and to make it work in conjunction with the neural networks.

3.5. User-provided Constraints

We can condition our models on any user-provided element constraints that we can encode into a sequence. We add an encoder to both the element constraint model and the edge model, following the encoder/decoder architecture described in [55]. The encoder takes as input a flattened sequence of user-provided constraints, enabling cross-attention from the sequence that is currently being generated to the list of user constraints. Note that the user-provided constraints do not have to represent the same quantities as the output sequence. In floor plans, for example, we can condition both the element constraint model and the edge model on a list of room types, room areas and/or a floor plan boundary.

3.6. Network Architecture

Our models use the Transformer [55] as a building block. Our Element Constraint Model and the Edge model are very similar to the Vertex and Face models from PolyGen [37] in organization. The building block for the Transformers themselves is based on the GPT-2 model, specifically, we use the GELU activation [14], Layer Norm [62] and Dropout. For a complete description, please refer to the supplementary.

We implemented our models in Pytorch[42]. Our models and sequences are small enough so we train on a single NVIDIA-V100 GPU with 32 GB memory. We use the Adam [26] optimizer, with a constant learning-rate of 10^{-4}, and linear warmup for 500 iterations. The element generation model is trained for 40 epochs, while the other models are trained for 80 epochs. It takes approximately 6 hours to train for our largest model for constrained generation.

The inference time depends on the type of sequence being sampled. Our large sequences have about 250 tokens. For this sequence length, generating a batch of 100 element constraint sequences takes about 10s. Given the element constraint sequence, all types of edges can be sampled in parallel. Edge models are larger and need about 60s for a batch of 100 sequences.

4. Results

We evaluate free generation of layouts, generation constrained by a given boundary, and generation constrained by additional user-provided constraints. We will focus on floor plans in this section. Furniture layouts are evaluated in the supplementary material.

Datasets We train and evaluate on two floor plan datasets. The RPLAN dataset [61] contains 80k floor plans of apartments or residential buildings in an Asian real estate market between 60m^2 to 120m^2. The LIFULL dataset [39] contains 61k floor plans of apartments from the Japanese housing market. The apartments in this dataset tend to be more compact. The original dataset is given as heterogeneous images,

Table 1. Free generation of layouts. We compare FID and layout statistics on two datasets to the state-of-the-art. Note that Graph2Plan uses a ground-truth layout graph as input, and both RPLAN and Graph2Plan use the ground truth boundary. We evaluate both free generation with our method and conditional generation. Our method improves upon the baselines with less input guidance.

dataset	method	FID	$\hat{s}_t$	$\hat{s}_r$	$\hat{s}_a$	$\hat{\mathbf{s}}_{\mathbf{avg}}$
RPLAN	StyleGAN.	25.29	46.74	4.41	7.85	19.67
	Graph2Plan	29.26	0.83	5.63	18.93	8.46
	RPLAN	**21.29**	5.38	1.53	4.38	3.76
	ours free	21.47	1.00	1.00	**1.00**	**1.00**
	ours cond.	27.27	**0.81**	**0.94**	1.34	1.03
LIFULL	StyleGAN	28.06	44.54	2.32	1.96	16.27
	Graph2Plan	29.50	9.21	0.94	1.37	3.84
	RPLAN	32.98	40.54	2.02	4.10	15.55
	ours free	**26.15**	1.00	1.00	1.00	**1.00**
	ours cond.	31.94	5.70	**0.71**	**0.50**	2.30

Table 2. Free generation comparison to HouseGAN on the subset of metrics that do not require door connectivity (denoted $\hat{s}^*$).

dataset	method	FID	$\hat{s}^*_t$	$\hat{s}_r$	$\hat{s}^*_a$	$\hat{\mathbf{s}}^*_{\mathbf{avg}}$
LIFULL	HouseGAN	35.58	4.01	3.66	3.50	3.72
	ours free	**26.15**	**1.00**	**1.00**	**1.00**	**1.00**

but a subset was parsed by Liu et al. [30] into a vector format. Even though both datasets contain apartment floor plans, the room layouts are quite different. Examples are shown in the last row of Figure 4. In both datasets we use 1k layouts for each of testing and validation, and the remainder for training.

Baselines StyleGAN [24] generates a purely image-based representation of a layout. We render the layout into an image to obtain a training set, including doors and walls (see the supplementary material), and parse the generated images to obtain layouts. Graph2Plan [16] generates a floor plan given its boundary and a layout graph that describes rough room locations, types, and adjacencies. Door connectivity is generated heuristically. RPLAN [61] generates a floor plan given its boundary, with a heuristically-generated door connectivity. HouseGAN [38] generates a floor plan given a room adjacency graph, but does not generate door connectivity. All baselines are re-trained on each dataset.

Metrics We compare generated layouts to ground truth layouts using two metrics: The *Fréchet Inception Distance* (FID) [15] computed on rendered layouts, and a metric based on a set of *layout statistics* that measure layout properties that the image-based FID is less suitable for. Layout statistics are grouped into topological statistics S_t such as the average graph distance in the layout graph between any two element types, element shape statistics S_r such as the aspect ratio or area, and alignment statistics S_a such as the gap between adjacent elements, or their boundary alignment. We believe that our proposed statistics are more useful to evaluate layouts than FID. FID is more suitable to evaluate generative models trained on natural images, but we show the FID metric for completeness as it is more widely used.

Topological statistics S_t are specialized to measure the topology of a layout graph [33, 52]:

- s^r_t: the average number of elements of a given type in a layout.
- s^h_t: a histogram over the number of elements of a given type in a layout.
- s^t_t: the number of connections between elements of type a and elements of type b in a layout.
- s^d_t: the average graph distance between elements of type a and elements of type b in a layout.
- s^e_t: a histogram of the graph distance from an element of type a to the exterior.
- s^c_t: a histogram of the degree of an element of type a, i.e. how many connections the element has to other elements.
- s^u_t: The number of inaccessible elements of type a in a layout.

Element shape statistics S_r measure simple properties of the element bounding boxes:

- s^c_r: a histogram of location distributions for each element type.
- s^a_r: a histogram of area distributions for each element type.
- s^s_r: a histogram of aspect ratio distributions for each element type.

Alignment statistics S_a measure alignment between all pairs of elements:

- s^c_a: a histogram of the distances between element centers, separately in x and y direction.
- s^g_a: a histogram of the gap size distribution between element bounding boxes (negatives values for overlaps).
- s^a_a: a histogram of the distances between element centers along the best-aligned (x or y) axis.
- s^s_a: a histogram of the distances between the best-aligned sides of the element bounding boxes.

The same alignment statistics are also computed between pairs of elements that are connected by descriptive edges.

We average each statistic over all layouts in a dataset and compare the resulting averages $\overline{s}$ to the statistics of the test set. We use the Earth Mover's distance [50] to compare histograms:

$$\hat{s}_* = \frac{1}{|S_*|} \sum_{s \in S_*} \frac{\text{EMD}(\overline{s}, \overline{s}^{\text{gt}})}{\text{EMD}(\overline{s}^{\text{ours}}, \overline{s}^{\text{gt}})}, \tag{5}$$

where $\overline{s}^{\text{ours}}$ and $\overline{s}^{\text{gt}}$ are the average statistics of our and ground truth distributions, and $*$ can be t, r or a. The average over all $\hat{s}_*$ is denoted $\hat{\mathbf{s}}_{\mathbf{avg}}$. Non-histogram statistics use the L2 distance instead of the EMD. The denominator normalizes the statistics, effectively giving the statistical error relative to our method. Thus our method will always have an error of 1 in this metric.

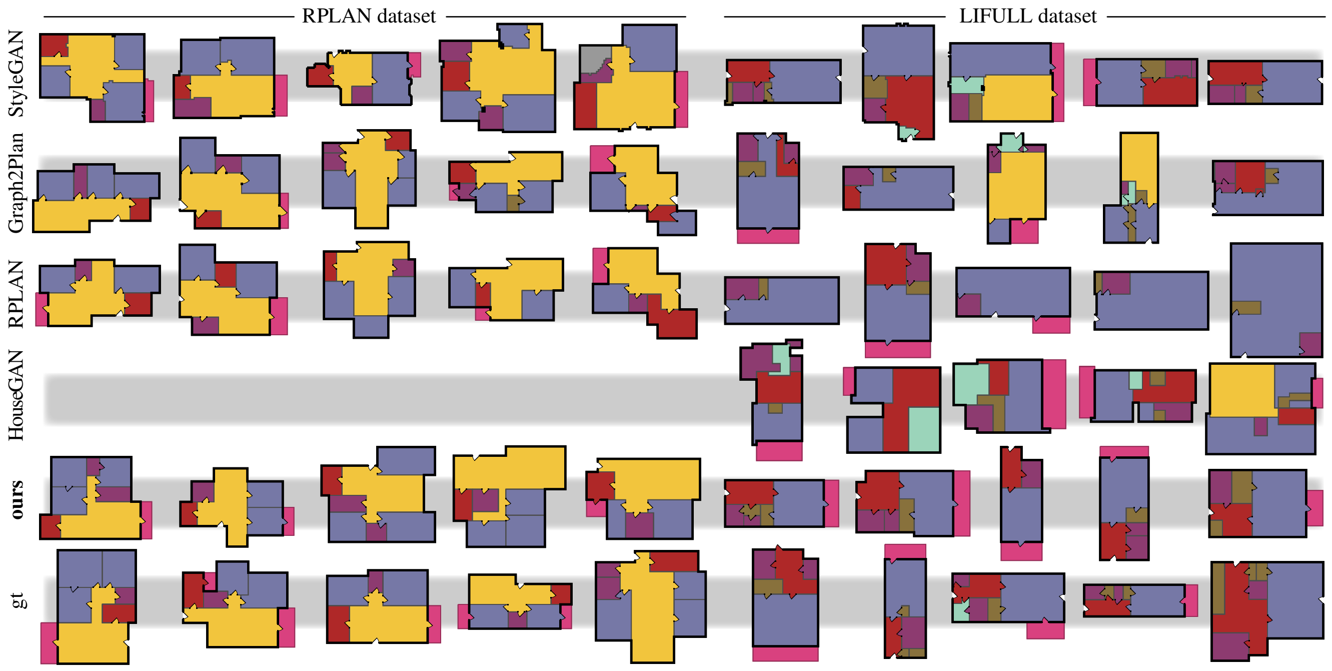
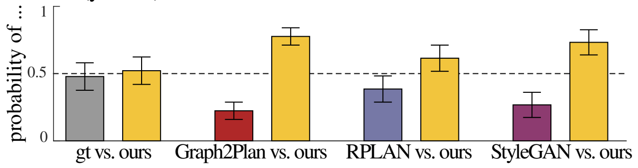


Figure 4. Free generation of floor plans. We compare our method to four baselines and to the ground truth datasets (gt). We show five unconditionally generated samples per dataset and per method. Rooms are colored by room type, doors connecting two rooms are shown as architectural door icons. Our three-step approach improves upon the room layout and connectivity compared to previous approaches, while requiring less guidance as input.

Free Generation First, we generate floor plans without any user input, by sampling the distribution learned by our constraint model. A comparison to all baselines is shown in Table 1 and Figure 4. The comparison to HouseGAN is shown in Table 2, where we only use the subset of statistics that are not based on the door connectivity, since HouseGAN does not create doors. Note that among the baselines, only StyleGAN can generate floor plans without user input, while Graph2Plan, RPLAN and HouseGAN need important parts of the ground truth as input. Providing this additional input gives Graph2Plan, RPLAN and HouseGAN a significant advantage in this comparison. The FID score correlates most strongly with the adjacency statistics, since adjacencies can be captured by only considering small spatial neighborhoods around corners and walls of a floor plan, but does not capture topology or room shape statics accurately that require considering larger-scale features. Unsurprisingly, StyleGAN performs reasonably well on the FID score and adjacency statistics, but shows a poor performance on topological statistics which are mainly based on larger-scale combinatorial features of the floor plans. Graph2Plan receives the topology as input giving it a good performance in topological statistics, but it struggles with room alignment. The RPLAN baseline is specialized to the RPLAN dataset, as shown in the large performance gap between RPLAN and LIFULL. HouseGAN does not generate doors, thus the topology statistics and alignment statistics are not directly comparable to the other methods. We can see that room shapes $\hat{s}_r$ are not as accurate as for the other baselines, and our method has improved performance in all statistics.

Table 3. Perceptual study. We show the probability of choosing our method (yellow) vs each baseline with 95% confidence intervals.



In summary, our framework improves significantly on the state-of-the art, in terms of layout topology, element shape, and element alignment, even though RPLAN and Graph2Plan received significant help from ground truth data.

Perceptual Study We conducted a two-alternative forced choice (2AFC) perceptual study to compare the perceived realism of the generated floor plans. Participants were shown top-down illustrations of two generated floor plans in the same style as Figure 1 and asked to pick the more realistic one. Each pair compares our result to a baseline. We used models trained on the RPLAN dataset, since our participants come from geographic regions where the RPLAN style of apartments is prevalent, and thus aligned with a participant's expectation. 10 users completed the study, each performing between 40 and 45 comparisons, for a total of 435 pairwise comparisons. Results are shown in Table 3. Note that our results are consistently rated more realistic than other methods with high confidence.

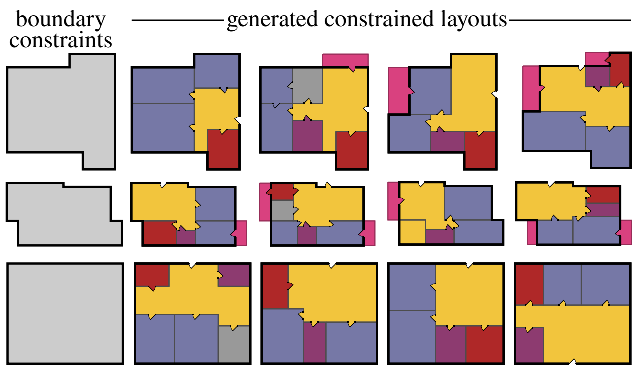
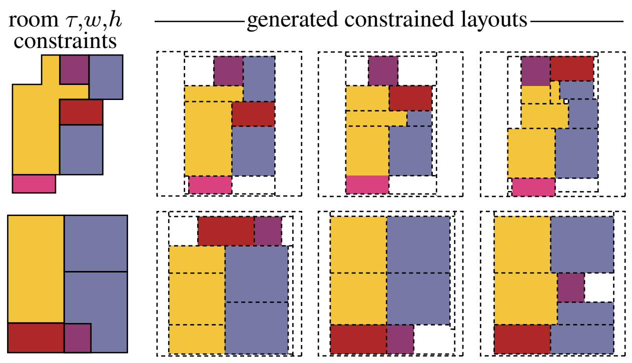


Figure 5. Boundary-constrained generation. Left: input boundary constraint; right: floor plans generated with this constraint.

Boundary-constrained Generation As described in Section 3.5, we can condition both our element constraint model and our edge model on input constraints provided by the user. Here, we show floor plan generation constrained by an exterior floor plan boundary given by the user. We parse the exterior of the given boundary into a sequence of rectangular elements that we use as input sequence for the encoders of our models. At training time, we use the exterior of ground truth floor plans as input. This trains the models to output sequences of element constraints and edges that are roughly compatible with the given boundary. In the optimization step, we add non-overlap constraints between the generated boxes and the given boundary. Additionally, since the interior boxes are generated in sequence from left to right, we can initialize the first generated box to match the left-most part of the interior area. Figure 5 show multiple examples of floor plans that were generated for the boundary given on the left. Quantitative results are provided in Table 1, under *ours cond*. These results were obtained by generating floor plans for all boundaries in the test set. The boundary-constrained floor plans show slightly lower performance in the average layout statistics and FID scores, but still perform much better than RPLAN, which also receives the boundary as input. We can see that our approach gives realistic floor plans that satisfy the given boundary constraint.

Element-constrained Generation Our approach can also handle constraints that are given in a different format than the output. We constrain our model to produce a given set of room types, widths, and heights. Results are shown in Figure 6. Even though these constraints are quite limiting, our model produces a large variety of results, while still approximately satisfying the given constraints. Similarly, we can constrain generation by given room counts or by the existence of a room type. Results are shown in Figure 7 .

Discussion Our work also has some limitations. For example, the constraint generation network can generate invalid constraints between elements, e.g. doors between rooms that do not share a wall. We can easily identify and remove these constraints. In addition, some constraints result in optimization problems that are infeasible. We simply ignore such samples. Further, like other methods, our work generates a small percentage of low quality results, however, significantly fewer than other methods, as shown in the statistics.


room τ,w,h constraints
generated constrained layouts
Figure 6. Element-constrained generation. Left: The type, width, and height of the these rooms are used as input constraints. Right: example layouts generated with these constraints. Note that the elements form regions of the same types and approximately the same width and height as the room constraints.

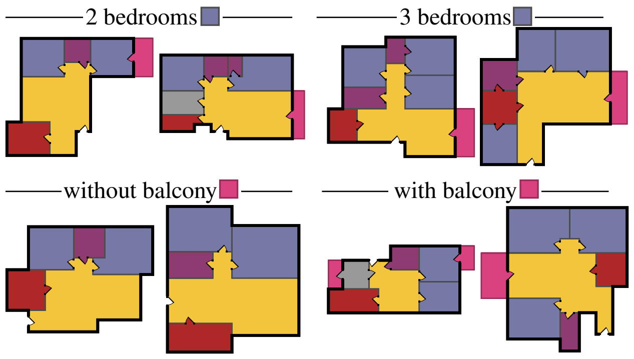


Figure 7. Generation constrained by room counts/existence. Constraints are described in the text above the samples. Our method can satisfy these more abstract constraints in most cases.

5. Conclusion

We proposed a new generative model for layouts. Our model first generates a layout graph with layout elements as nodes and constraints between layout elements as edges. The final layout is computed by optimization. Our model overcomes many limitations of previous models, mainly the need for significant user input and ad-hoc post-processing steps. Further, our model leads to significantly higher generation quality as evidenced by multiple statistics and enables multiple possibilities of conditional layout generation. In future work, we would like to explore the application of our model to other layout problems, such as image layouts, 3D scene layouts, and component-based object modeling. We also would like to explore if our model can be used to post-process 3D scans of indoor environments.

References

[1] Oron Ashual and Lior Wolf. Specifying object attributes and relations in interactive scene generation. *International Conference on Computer Vision*, 2019.

[2] Fan Bao, Dong-Ming Yan, Niloy J. Mitra, and Peter Wonka. Generating and exploring good building layouts. *ACM Transactions on Graphics*, 32(4), 2013.

[3] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[4] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale GAN training for high fidelity natural image synthesis. 2018.

[5] Stanislas Chaillou. Archigan: Artificial intelligence x architecture. In *Architectural Intelligence*, pages 117–127. Springer, 2020.

[6] Siddhartha Chaudhuri, Evangelos Kalogerakis, Leonidas Guibas, and Vladlen Koltun. Probabilistic reasoning for assembly-based 3D modeling. *ACM Transactions on Graphics*, 2011.

[7] Mark Chen, Alec Radford, Rewon Child, Jeff Wu, Heewoo Jun, Prafulla Dhariwal, David Luan, and Ilya Sutskever. Generative pretraining from pixels. In *Proceedings of the 37th International Conference on Machine Learning*, 2020.

[8] Mark Chen, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, and Ilya Sutskever. Generative pretraining from pixels. In *International Conference on Machine Learning*, 2020.

[9] Hang Chu, Daiqing Li, David Acuna, Amlan Kar, Maria Shugrina, Xinkai Wei, Ming-Yu Liu, Antonio Torralba, and Sanja Fidler. Neural turtle graphics for modeling city road layouts. *International Conference on Computer Vision*, 2019.

[10] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[11] Lubin Fan and Peter Wonka. A probabilistic model for exteriors of residential buildings. *ACM Transactions on Graphics*, 2016.

[12] Matthew Fisher, Daniel Ritchie, Manolis Savva, Thomas Funkhouser, and Pat Hanrahan. Example-based synthesis of 3d object arrangements. *ACM Transactions on Graphics*, 2012.

[13] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, 2014.

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

[15] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *Advances in Neural Information Processing Systems*, 2017.

[16] Ruizhen Hu, Zeyu Huang, Yuhan Tang, Oliver Van Kaick, Hao Zhang, and Hui Huang. Graph2plan: Learning floorplan generation from layout graphs. *Proceedings of SIGGRAPH*, 2020.

[17] Xun Huang, Ming-Yu Liu, Serge J. Belongie, and Jan Kautz. Multimodal unsupervised image-to-image translation. *ECCV 2018*, 2018.

[18] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. *arxiv*, 2016.

[19] Justin Johnson, Agrim Gupta, and Li Fei-Fei. Image generation from scene graphs. *Conference on Computer Vision and Pattern Recognition*, 2018.

[20] Akash Abdu Jyothi, Thibaut Durand, Jiawei He, Leonid Sigal, and Greg Mori. Layoutvae: Stochastic scene layout generation from a label set. *International Conference on Computer Vision*, 2019.

[21] Evangelos Kalogerakis, Siddhartha Chaudhuri, Daphne Koller, and Vladlen Koltun. A probabilistic model for component-based shape synthesis. *ACM Transactions on Graphics*, 2012.

[22] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *CoRR*, 2017.

[23] Tero Karras, Miika Aittala, Janne Hellsten, Samuli Laine, Jaakko Lehtinen, and Timo Aila. Training generative adversarial networks with limited data. In *Advances in Neural Information Processing Systems*, 2020.

[24] Tero Karras, Samuli Laine, and Timo Aila. A Style-Based Generator Architecture for Generative Adversarial Networks. *arXiv e-prints*, 2018.

[25] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan. *arXiv*, 2019.

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

[27] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[28] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. In *ICLR*, 2018.

[29] Renjie Liao, Yujia Li, Yang Song, Shenlong Wang, Will Hamilton, David K Duvenaud, Raquel Urtasun, and Richard Zemel. Efficient graph generation with graph recurrent attention networks. In *Advances in Neural Information Processing Systems*, 2019.

[30] Chen Liu, Jiajun Wu, Pushmeet Kohli, and Yasutaka Furukawa. Raster-to-vector: Revisiting floorplan transformation. *International Conference on Computer Vision*, 2017.

[31] Jenny Liu, Aviral Kumar, Jimmy Ba, Jamie Kiros, and Kevin Swersky. Graph normalizing flows. In *Advances in Neural Information Processing Systems*, 2019.

[32] L. Majerowicz, A. Shamir, A. Sheffer, and H. H. Hoos. Filling your shelves: Synthesizing diverse style-preserving artifact arrangements. *IEEE Transactions on Visualization and Computer Graphics*, 2014.

[33] S. Marshall. *Streets and Patterns*. Routledge, 2015.

[34] Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-generated residential building layouts. *ACM Transactions on Graphics*, 2010.

[35] Paul Merrell, Eric Schkufza, Zeyang Li, Maneesh Agrawala, and Vladlen Koltun. Interactive furniture layout using interior design guidelines. *ACM Transactions on Graphics*, 2011.

[36] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Transactions on Graphics*, 2006.

[37] Charlie Nash, Yaroslav Ganin, SM Eslami, and Peter W Battaglia. Polygen: An autoregressive generative model of 3d meshes. *arXiv preprint arXiv:2002.10880*, 2020.

[38] Nelson Nauata, Kai-Hung Chang, Chin-Yi Cheng, Greg Mori, and Yasutaka Furukawa. House-gan: Relational generative adversarial networks for graph-constrained house layout generation. 2020.

[39] National Institute of Informatics. LIFULL HOME'S Dataset, 2020.

[40] S. Pan, R. Hu, S. Fung, G. Long, J. Jiang, and C. Zhang. Learning graph embedding with adversarial training methods. *IEEE Transactions on Cybernetics*, 2020.

[41] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Łukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. *arXiv preprint arXiv:1802.05751*, 2018.

[42] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[43] Chi-Han Peng, Yong-Liang Yang, Fan Bao, Daniel Fink, Dong-Ming Yan, Peter Wonka, and Niloy J Mitra. Computational network design from functional specifications. *ACM Transactions on Graphics*, 2016.

[44] Chi-Han Peng, Yong-Liang Yang, and Peter Wonka. Computing layouts with deformable templates. *ACM Transactions on Graphics*, 2014.

[45] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990.

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

[47] Ali Razavi, Aäron van den Oord, and Oriol Vinyals. Generating diverse high-fidelity images with VQ-VAE-2. In *Advances in Neural Information Processing Systems*, 2019.

[48] Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. *arXiv preprint arXiv:1505.05770*, 2015.

[49] Elad Richardson, Yuval Alaluf, Or Patashnik, Yotam Nitzan, Yaniv Azar, Stav Shapiro, and Daniel Cohen-Or. Encoding in style: a stylegan encoder for image-to-image translation. *arXiv preprint arXiv:2008.00951*, 2020.

[50] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. A metric for distributions with applications to image databases. In *International Conference on Computer Vision*, 1998.

[51] Martin Simonovsky and Nikos Komodakis. GraphVAE: towards generation of small graphs using variational autoencoders. In *ICLR*, 2018.

[52] Sherif Tarabishy, Stamatios Psarras, Marcin Kosicki, and Martha Tsigkari. Deep learning surrogate models for spatial and visual connectivity. *ArXiv*, 2019.

[53] Arash Vahdat and Jan Kautz. NVAE: A deep hierarchical variational autoencoder. In *Advances in Neural Information Processing Systems*, 2020.

[54] Carlos A. Vanegas, Tom Kelly, Basil Weber, Jan Halatsch, Daniel Aliaga, and Pascal Müller. Procedural generation of parcels in urban modeling. *Computer Graphics Forum*, 2012.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

[56] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, 2015.

[57] Hongwei Wang, Jia Wang, Jialin Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Xing Xie, and Minyi Guo. Graphgan: Graph representation learning with generative adversarial nets. In *AAAI*, 2018.

[58] Kai Wang, Yu-An Lin, Ben Weissmann, Manolis Savva, Angel X. Chang, and Daniel Ritchie. Planit: planning and instantiating indoor scenes with relation graph and spatial prior networks. *ACM Transactions on Graphics*, 2019.

[59] Kai Wang, Manolis Savva, Angel X. Chang, and Daniel Ritchie. Deep convolutional priors for indoor scene synthesis. *ACM Transactions on Graphics*, 2018.

[60] Wenming Wu, Lubin Fan, Ligang Liu, and Peter Wonka. Miqp-based layout design for building interiors. *Computer Graphics Forum*, 2018.

[61] Wenming Wu, Xiao-Ming Fu, Rui Tang, Yuhan Wang, Yu-Hao Qi, and Ligang Liu. Data-driven interior plan generation for residential buildings. *ACM Transactions on Graphics*, 2019.

[62] Jingjing Xu, Xu Sun, Zhiyuan Zhang, Guangxiang Zhao, and Junyang Lin. Understanding and improving layer normalization. In *Advances in Neural Information Processing Systems*, 2019.

[63] Guandao Yang, Xun Huang, Zekun Hao, Ming-Yu Liu, Serge Belongie, and Bharath Hariharan. Pointflow: 3d point cloud generation with continuous normalizing flows. In *Proceedings of the IEEE International Conference on Computer Vision*, 2019.

[64] Yong-Liang Yang, Jun Wang, Etienne Vouga, and Peter Wonka. Urban pattern: Layout design by hierarchical domain splitting. *ACM Transactions on Graphics*, 2013.

[65] Yi-Ting Yeh, Katherine Breeden, Lingfeng Yang, Matthew Fisher, and Pat Hanrahan. Synthesis of tiled patterns using factor graphs. *ACM Transactions on Graphics*, 2013.

[66] Yi-Ting Yeh, Lingfeng Yang, Matthew Watson, Noah D. Goodman, and Pat Hanrahan. Synthesizing open worlds with constraints using locally annealed reversible jump mcmc. *ACM Transactions on Graphics*, 2012.

[67] Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. GraphRNN: generating realistic graphs with deep auto-regressive models. In *ICML*, 2018.

[68] Lap-Fai Yu, Sai-Kit Yeung, Chi-Keung Tang, Demetri Terzopoulos, Tony F. Chan, and Stanley J. Osher. Make it home: Automatic optimization of furniture arrangement. *ACM Transactions on Graphics*, 2011.

[69] Lap-Fai Yu, Sai-Kit Yeung, Chi-Keung Tang, Demetri Terzopoulos, Tony F. Chan, and Stanley J. Osher. Make it home: Automatic optimization of furniture arrangement. *ACM Transactions on Graphics*, 2011.

[70] Han Zhang, Ian J. Goodfellow, Dimitris N. Metaxas, and Augustus Odena. Self-attention generative adversarial networks. *arXiv:1805.08318*, 2018.

[71] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017.

[72] Jun-Yan Zhu, Richard Zhang, Deepak Pathak, Trevor Darrell, Alexei A Efros, Oliver Wang, and Eli Shechtman. Toward multimodal image-to-image translation. In *Advances in Neural Information Processing Systems*. 2017.

[73] Peihao Zhu, Rameen Abdal, Yipeng Qin, and Peter Wonka. Sean: Image synthesis with semantic region-adaptive normalization. In *Conference on Computer Vision and Pattern Recognition*, 2020.