PlanIT: Planning and Instantiating Indoor Scenes with Relation Graph and Spatial Prior Networks

KAI WANG, Brown University
YU-AN LIN, Brown University
BEN WEISSMANN, Brown University
MANOLIS SAVVA, Simon Fraser University
ANGEL X. CHANG, Simon Fraser University
DANIEL RITCHIE, Brown University

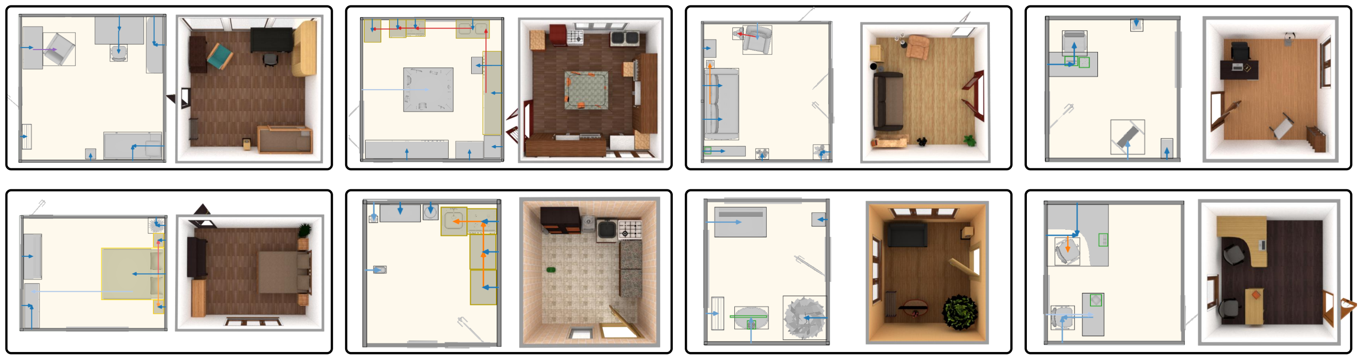


Fig. 1. We present PlanIT: a scene synthesis framework that unifies high-level *planning* of scene structure using a generative model over relation graphs with lower-level spatial *instantiation* using neural-guided search with spatial neural network modules. We first generate a relation graph with objects at the nodes and spatial or semantic relations at the edges (left images). Then, given the graph structure we select and place objects to instantiate a concrete 3D scene.

We present a new framework for interior scene synthesis that combines a high-level relation graph representation with spatial prior neural networks. We observe that prior work on scene synthesis is divided into two camps: object-oriented approaches (which reason about the set of objects in a scene and their configurations) and space-oriented approaches (which reason about what objects occupy what regions of space). Our insight is that the object-oriented paradigm excels at high-level *planning* of how a room should be laid out, while the space-oriented paradigm performs well at *instantiating* a layout by placing objects in precise spatial configurations. With this in mind, we present PlanIT, a layout-generation framework that divides the problem into two distinct *planning* and *instantiation* phases. PlanIT represents the "plan" for a scene via a relation graph, encoding objects as nodes and spatial/semantic relationships between objects as edges. In the planning phase, it uses a deep graph convolutional generative model to synthesize relation graphs. In the instantiation phase, it uses image-based convolutional network modules to guide a search procedure that places objects into the scene in a manner consistent with the graph. By decomposing the problem in this way, PlanIT generates scenes of comparable quality to those generated by prior approaches (as judged by both people and learned classifiers), while also providing the modeling flexibility of the intermediate relationship graph representation. These graphs allow the system to support applications such as scene synthesis from a partial graph provided by a user.

Authors' addresses: Kai Wang, Brown University; Yu-an Lin, Brown University; Ben Weissmann, Brown University; Manolis Savva, Simon Fraser University; Angel X. Chang, Simon Fraser University; Daniel Ritchie, Brown University.

CCS Concepts: • **Computing methodologies** → **Computer graphics**; **Neural networks**; **Probabilistic reasoning**;

Additional Key Words and Phrases: indoor scene synthesis, object layout, neural networks, convolutional networks, deep learning, relationship graphs, graph generation

ACM Reference Format:
Kai Wang, Yu-an Lin, Ben Weissmann, Manolis Savva, Angel X. Chang, and Daniel Ritchie. 2019. PlanIT: Planning and Instantiating Indoor Scenes with Relation Graph and Spatial Prior Networks. *ACM Trans. Graph.* 38, 4, Article 132 (July 2019), 15 pages. https://doi.org/10.1145/3306346.3322941

1 INTRODUCTION

People spend a large percentage of their lives indoors—in bedrooms, living rooms, kitchens, etc. As computer graphics reproduces the real world in increasing fidelity, the demand for virtual versions of such spaces also grows. Virtual and augmented reality experiences often take place in such environments. Online virtual interior design tools are available to help people redesign their own spaces [Planner5d 2017; RoomSketcher 2017]. Some furniture design companies now primarily advertise their products by rendering virtual scenes, as it is faster, cheaper, and more flexible to do so than to stage real-world scenes [Chaos Group 2018]. Finally, machine learning researchers have begun turning to virtual environments to train data-hungry models for computer vision and robotic navigation [Dai et al. 2018; Das et al. 2018; Gordon et al. 2018; Lange 2018].

Given this interest in virtual indoor scenes and the large amount of effort required to author them with traditional tools, a generative model of such scenes would be valuable. Indeed, many researchers have studied this problem of *indoor scene synthesis* over the past decade. Two major families of approaches have emerged in this line of work. The first family of models is *object-oriented*: they explicitly represent the set of objects in the scene and their properties [Fisher et al. 2012; Li et al. 2018a; Qi et al. 2018; Zhang et al. 2018]. The second family of models is *space-oriented*: they instead treat space as a first-class entity, modeling what occupies each point in space. Here, space typically takes the form of a regular grid, as in recent image-based scene synthesis models [Ritchie et al. 2019; Wang et al. 2018]. This modeling dichotomy is analogous to the division between Lagrangian and Eulerian simulation methods.

Each approach has strengths and weaknesses. The object-oriented paradigm facilitates explicit reasoning about objects as discrete entities, supporting symbolic queries such as "generate a scene with a chair and two tables." This representation also facilitates detecting and exploiting high-level arrangement patterns, such as symmetries. However, because object-oriented approaches abstract away low-level spatial details (such as the precise geometry of the objects and the architecture of rooms), they can struggle with fine-grained arrangement. Space-oriented systems, by contrast, excel at complex low-level spatial reasoning (supporting arbitrarily shaped rooms and irregular object geometry) but often miss high-level patterns and do not support symbolic queries.

In this paper, we propose **PlanIT**, a new conceptual framework for layout generation that unites the object-oriented and space-oriented paradigms to achieve the best of both worlds. Specifically, PlanIT *plans* the structure of a scene by generating a relationship graph, and then it *instantiates* a concrete 3D scene which conforms to that plan. Relationship graphs help our system reason symbolically about objects and their high-level patterns. In a relationship graph, each node represents an object, and each directed edge represents a spatial or functional relationship between two objects. We learn a generative model of such graphs which can be sampled to synthesize new high-level scene layouts. Then, given a graph, we use a set of image-based (i.e. space-oriented) models to *instantiate* the high-level relationship graph into a low-level collection of arranged 3D objects. Each part of the system focuses on the domain at which it excels: the graph-based module reasons about which objects should be in the scene and their high-level arrangement patterns, while the image-based modules determines precise placements, orientations, and object sizes. This pipeline can be viewed as an abstraction hierarchy: we first synthesize an abstract scene (i.e. a graph), and then we synthesize a concrete scene conditioned on the abstract representation. We believe that this multi-resolution modeling approach is a better match to the nature of real scenes (high-level structural variation in arrangements, and low-level details in objects) and also mirrors the human thought process when designing and laying out spaces (starting from overall object layouts in a room, and then placing individual objects). Moreover, the graph is a useful intermediate representation for many applications that involve composition, editing, and manipulation of scene structure. While we focus on the domain of indoor scene synthesis in this paper, we believe that PlanIT's "plan-and-instantiate" framework has broader applicability to other layout-generation problems in computer graphics.

Our system starts with a large set of unstructured 3D scenes and automatically extracts relationship graphs using geometric and statistical heuristics. To instantiate these graphs into concrete scenes, we use a neurally-guided search procedure building upon convolutional neural network (CNN)-based scene synthesis modules from recent work [Ritchie et al. 2019]. To learn how to generate the graphs themselves, we again leverage convolution as an operator for capturing context. However, instead of spatial context (in the form of image neighborhoods), we capture symbolic relational context via graph convolutional networks (GCN). We use a GCN-based generative model of scene relationship graphs which builds on recent work on deep generative models of graphs [Li et al. 2018b]. Our pipeline can be used to synthesize new scenes from scratch, to complete partial scenes, or to synthesize scenes from a complete or partial graph specification. The latter paradigm has applications both for rapid, accessible scene design as well as for generating custom-tailored training environments for vision-based autonomous agents.

Our unified scene synthesis pipeline generates scenes that are judged (by both learned classifiers and people) to be of comparable quality to scenes generated by prior methods that are either space-oriented or object-oriented. We also demonstrate that our pipeline enables applications such as generating 3D scenes from partially specified scene graphs, and generating custom 3D scenes for training robotics and vision systems.

In summary, our contributions are:

(1) A novel "plan-and-instantiate" conceptual framework for layout generation problems, and a concrete implementation of this framework for the domain of indoor scene synthesis.
(2) A formulation of relationship graphs for indoor scene layouts and a heuristic procedure for extracting them from unstructured 3D scenes.
(3) An introduction to deep generative graph models based on message-passing graph convolution, and a specific model architecture for generating indoor scene relationship graphs.
(4) A neurally-guided search procedure for instantiating scene relationship graphs, using image-based scene synthesis models augmented with an awareness of the input graph.

Source code and pretrained models for our system can be found at https://github.com/brownvc/planit.

2 BACKGROUND & RELATED WORK

We discuss related work in scene synthesis, scene graph representations for related problems, and graph generative models.

Indoor Scene Synthesis: There is a long line of work addressing 3D scene synthesis. Early work used a rule-based constraint satisfaction formulation to generate 3D object layouts for pre-specified sets of objects [Xu et al. 2002]. Other approaches were based on optimization of cost functions derived from interior design principles [Merrell et al. 2011] and object–object statistical relationships [Yu et al. 2011]. The earliest data-driven work modeled object co-occurrences using a Bayesian network, and Gaussian mixtures for pairwise spatial relation statistics extracted from 3D scenes [Fisher et al. 2012]. Followup work has used undirected factor graphs learned from annotated RGB-D images [Kermani et al. 2016], relation graphs between objects learned from human activity annotations [Fu et al. 2017], and directed graphical models with Gaussian mixtures for modeling arrangement patterns [Paul Henderson 2018]. Other work has focused on conditioning the scene generation using input from RGB-D frames [Chen et al. 2014], 2D sketches of the scene [Xu et al. 2013], natural language text [Chang et al. 2015; Ma et al. 2018b], or activity predictions on RGB-D reconstructions [Fisher et al. 2015].

More recently, with the availability of large datasets of 3D environments such as SUNCG [Song et al. 2017], learning-based approaches have become popular. A variety of approaches have been proposed using: human-centric probabilistic grammars [Qi et al. 2018], Generative Adversarial Networks trained on a matrix representation of present scene objects [Zhang et al. 2018], recursive neural networks trained to sample 3D scene hierarchies [Li et al. 2018a], and convolutional neural networks (CNNs) trained on top-down image representations of rooms [Ritchie et al. 2019; Wang et al. 2018]. Our system uses the fast CNN modules of the latter image-based method to instantiate relationship graphs, modified significantly to work with a relationship graph as input.

Scene Graph Representations: Representing scenes as graphs of semantic relationships (encoded as edges) between objects (encoded as nodes) is an elegant methodology with applications to a variety of domains. It has been used for text-to-scene generation [Chang et al. 2014]. Other work in graphics has used a small dataset of manually annotated scene hierarchies to learn a grammar for predicting hierarchical decompositions of input 3D scenes [Liu et al. 2014]. In computer vision, semantic scene graphs were popularized by the Visual Genome project which collected a large dataset of image scene graph annotations [Krishna et al. 2017]. Subsequent work has focused on generating scene graphs from images [Li et al. 2017; Lu et al. 2016; Xu et al. 2017; Yang et al. 2018; Zellers et al. 2018], using scene graphs for image retrieval [Johnson et al. 2015], generating 2D images given an input scene graph [Johnson et al. 2018], and improving the evaluation of image captioning [Anderson et al. 2016]. This diversity of applications is enabled by the generality of the scene graph representation, and is one of our main motivations for incorporating a graph representation in our approach. In contrast to prior work, we focus on demonstrating that our unified formulation that combines a relation graph generative model with image-based neural networks enables us to improve 3D scene synthesis.

Graph Generative Models: Recently, there is an exciting line of work on defining graph–structured neural networks and using them to learn a generative models of graphs. Some of the earliest work in graph processing with neural networks proposed a graph convolutional neural network (GCN) to perform graph classification [Kipf and Welling 2016]. Other work has used a GCN approach for skeleton-based action recognition [Yan et al. 2018]. Our graph generative model uses a formulation of GCNs based on *message passing* between nodes in a graph [Gilmer et al. 2017]. It specifically builds on a recently-developed autoregressive generative model for arbitrary graphs using the message-passing approach [Li et al. 2018b]. Other recent work proposes a recurrent architecture for graphs, targeted at applications involving large network graphs [You et al. 2018b]. Our overall scene synthesis approach is based on combining a GCN-based generative model over 3D scene relationship graphs with image–based CNN modules. To our knowledge, we are the first to apply a deep generative graph model in computer graphics in general and to 3D scene synthesis in particular.

3 OVERVIEW

In this paper, we tackle the *scene synthesis* problem: given the architectural specification of a room (walls, floor, and ceiling) of a particular type (e.g. bedroom), choose and arrange a set of objects to create a plausible instance of that type of room. In doing so, we aim to build a system that can support a range of use cases. In addition to synthesizing a scene from an empty room, it should also complete partial scenes. Furthermore, it should accept a high-level specification for what should be in the room in the form of a partial or complete relation graph. For example, our system should support the query "give me a bedroom with a desk and a television, where the desk is to the left of the bed." Our approach to the problem is to decompose it into two steps. First, our system generates a relation graph. This graph encodes major salient relationships that characterize a scene layout but does not completely specify a scene. However, it does provide a strong signal from which to generate one or more instantiations of the graph.

We start by automatically extracting relation graphs from unstructured 3D scenes, using geometric rules and the statistics of a large database of scenes to decide what relationships exist and which ones are most salient (Figure 2 Left). Section 4 describes this process, as well as our graph representation, in more detail.

Next, we use this corpus of extracted graphs to learn a generative model of graphs (Figure 2 Middle). Our generative model takes an input graph that is either empty (i.e. containing only nodes and edges representing room architecture features such as walls) or partially occupied with objects, and generates additional nodes and edges to complete the graph. Our model is a deep generative model that uses a form of discrete convolution on graphs as its primary operator. It is based on an architecture which was applied to generate simple graphs in other domains, e.g. molecule structures [Li et al. 2018b]. Section 5 describes this model in more detail.

With a complete graph in hand, the final step in our pipeline is to *instantiate* the abstract graph into a concrete scene by choosing and placing 3D models which respect the objects and relationships implied by the graph (Figure 2 Right). There are many possible methods one could use to search for concrete scene layouts consistent with a relation graph. Since the scene layout process involves low-level spatial reasoning, we opt to use image-based neural network modules to guide our search. Using neural networks provides robustness to the noise present in relation graphs both in the training data and for the output graphs of our generative model. We adapt modules from recent work on scene synthesis [Ritchie et al. 2019], modifying them to take the graph as input and to attempt to adhere to the structure that it mandates. Section 6 describes these modules, and our overall scene instantiation search procedure, in more detail.

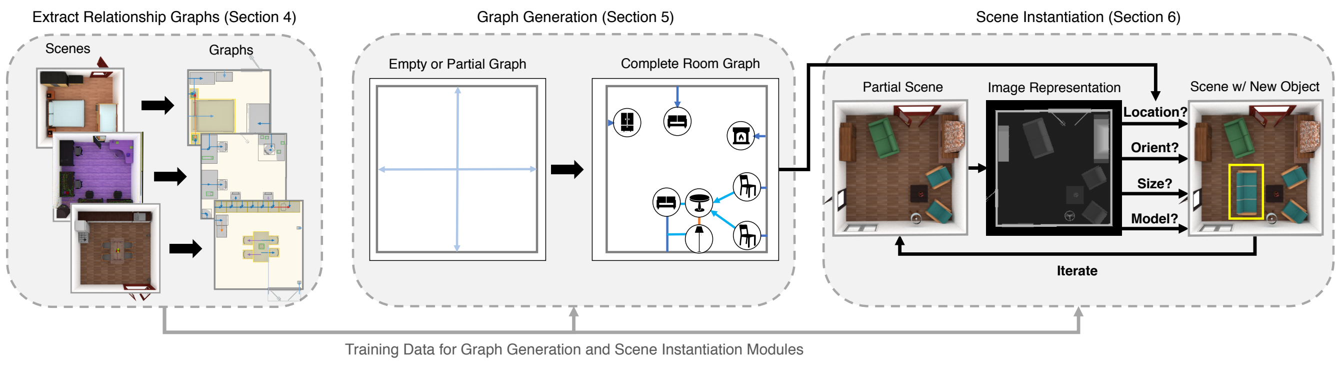

Extract Relationship Graphs (Section 4)
Scenes
Graphs
Graph Generation (Section 5)
Empty or Partial Graph
Complete Room Graph
Scene Instantiation (Section 6)
Partial Scene
Image Representation
Scene w/ New Object
Location?
Orient?
Size?
Model?
Iterate
Training Data for Graph Generation and Scene Instantiation Modules

Fig. 2. Our scene synthesis pipeline. We automatically extract relation graphs from scenes (Section 4), which we use to train a deep generative model of such graphs (Section 5). In this figure, the graph nodes are labeled with an icon indicating their object category. We then use image-based reasoning to *instantiate* a graph into a concrete scene by iterative insertion of 3D models for each node (Section 6).

4 TURNING SCENES INTO RELATION GRAPHS

In this section, we motivate and define our relation graph representation, and we describe a procedure for automatically extracting these graphs from unstructured 3D scenes.

4.1 Dataset

For all of our experiments, we use the SUNCG dataset, a collection of over forty thousand scenes designed by users of an online interior design tool [Song et al. 2017]. From this raw scene collection, we extract rooms of five common types: bedrooms, living rooms, bathrooms, and offices. We also perform pre-processing on these rooms to filter out mislabeled rooms, remove uncommon objects, etc., using a procedure based on one from prior work [Ritchie et al. 2019]. Our filtering procedure additionally removes rooms that are not closed (i.e. are not encircled by a closed loop of walls), as our graph representation requires this to be the case (Section 4.2). This results in 5900 bedrooms (with 41 unique object categories), 1100 living rooms (37 categories), 6400 bathrooms (26 categories), 1000 offices (37 categories), and 1900 kitchens (39 categories). These rooms are represented as a flat list of objects (with category label, geometry, and transformation); they contain no information about structural, functional, or semantic relationships between objects.

4.2 Graph Representation

We encode relationships between objects in the form of a directed relation graph. In this graph, each node denotes an object, and each edge encodes a directed spatial or functional relationship from one object to another. Each node is labeled with an object category (e.g. *wardrobe*) and a *functional symmetry type*. Functional symmetry refers to an object having multiple semantically-meaningful “front” directions, e.g. an L-shaped sectional sofa has two potential “front” directions, regardless of whether the sofa has a precise geometric symmetry). This determines the configurations in which the object can be placed while still serving the same function. Figure 3 shows examples of the symmetry types we capture in our graphs.

Relationship Edges: Edges in the relation graph capture important constraints between objects: that two objects must be related in some way due to physical laws or functional use. For physical plausibility, our graphs include **support** edges: edge $A \longrightarrow B$ implies that object B is physically supported by object A (e.g. a lamp supported by a table). To capture arrangement patterns that reflect functional use, graphs also include **spatial** edges: $A \longrightarrow B$ implies that object B is placed at some distance away from object A, in some direction relative to object A (e.g. an ottoman in front of a chair). We further subdivide spatial edges into multiple subtypes, defined as the Cartesian product of four direction types (**front**, **back**, **right**, **left**) and three distance types (**adjacent**, **proximal**, **distant**) for a total of 12 spatial edge types. Directions are defined in the local coordinate frame of the edge start node. We use discrete distance types because people use such terms when describing spatial relationships, suggesting that there may be salient categorical differences between different distance levels. Figure 4 shows some examples of different relationship types.

Representing the room architecture: The architectural geometry of the room influences the layout of objects within it and thus must also be modeled in the relation graph. We represent this geometry with additional nodes, one for each linear wall segment, connected by a bi-directional loop of *adjacent* edges (a bi-directional edge pair represents a conceptually undirected edge). Nodes for walls on opposite sides of the room are also connected to further encode the room shape in the graph structure. Wall nodes also store the length of the wall segment, and wall $\longrightarrow$ wall edges carry an additional attribute for the angle between the two adjacent walls. Doors and windows are represented as nodes adjacent to their respective wall(s). We do not include a floor node because it adds no meaningful information (any object without a supporting node is on the floor) and it over-connects the graph (every floor-supported object is just two hops away from every wall).

4.3 Graph Extraction

To convert a scene into the above graph representation, we use geometric heuristics to extract a superset of possible relationships that may exist between objects. We then use additional geometric

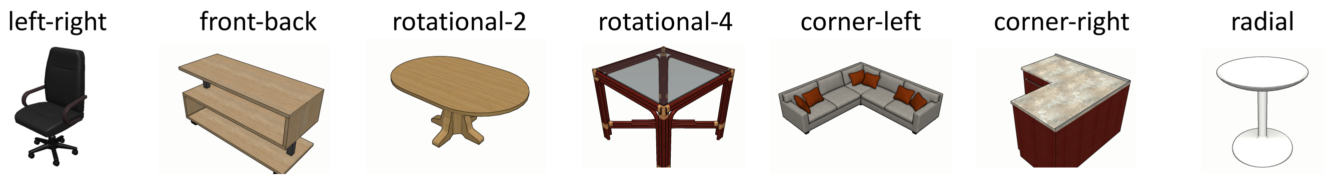


Fig. 3. Possible types of functional symmetries with which graph nodes can be labeled, along with an example object of that symmetry type.

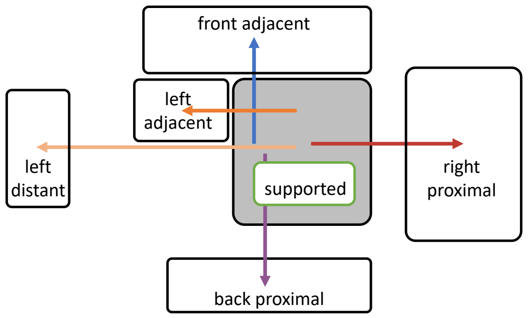


Fig. 4. Relationships modeled by the edges in our relation graphs. We define *support* edges for statically supported child nodes, and the four spatial edges *front*, *left*, *right*, and *back*, at three distances: *adjacent*, *proximal* and *distant*. The hue of each arrow indicates the relationship direction, and the saturation indicates distance. Supported nodes are outlined in green. We use this same color scheme throughout all figures in this paper.

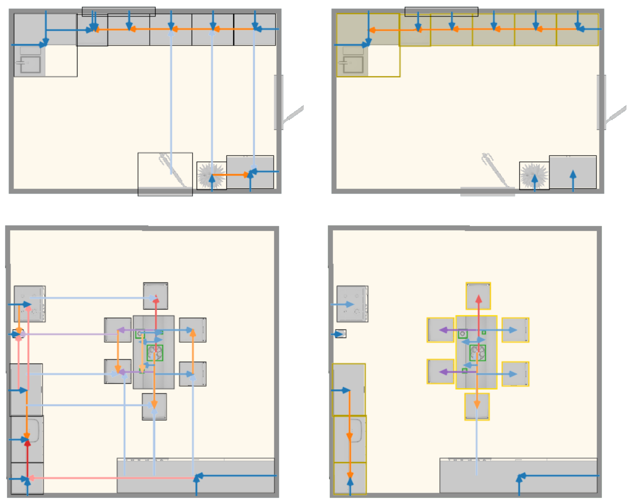


Fig. 5. Graphs before *(left)* and after *(right)* the detection of superstructures. Hub-and-spoke and chain superstructures are indicated with yellow and brown bounding boxes around member nodes, respectively. Superstructures organize relationships more compactly (e.g. the chains of kitchen cabinets along the walls and the chairs relative to the dining table).

and statistical heuristics to refine this set to reflect only the most meaningful relationships.

Functional symmetries: Each object node requires a functional symmetry type label. We manually label all 3D models in the SUNCG dataset, as the number of models in the dataset is not prohibitively large (~ 2600 models across all categories).

Support edges: For each object, we identify potential supporting parent objects by tracing a ray outwards from the bottom of the object's bounding box up to a threshold distance of 10 cm. If there are multiple potential supporting objects, we select the object that has the largest supporting surface. We break support edge cycles by unparenting the largest object in the cycle.

Spatial edges: We check for each possible direction of a spatial edge $A \longrightarrow B$ by raycasting from the four sides of the oriented bounding box (OBB) of A (projected into the XY plane). For an intersected object B to contribute an edge to the graph, at least 30% of the object must be visible from A (determined by the interval overlap of B's OBB onto A's OBB). If this condition is satisfied for multiple directions between A and B, we pick the one with the highest visibility. Since radially symmetric objects have no meaningful orientation, all relationships in which A is radially-symmetric are given the label *front*. Distance labels are determined by the distance between the two objects' OBBs: *adjacent* if A is within two inches of B or within 5% of the largest diagonal of the two objects (whichever is smaller), *proximal* if A is within 1.5 feet or 10% of the largest diagonal (whichever is larger), and *distant* otherwise.

Detecting "superstructures": Indoor scenes often contain functional groups of objects; our graphs should contain these structures so that generative models can learn to capture them. These groups can be detected by searching for "superstructure" patterns in the extracted relation graph.

We detect two types of such superstructures in our graphs: *hub-and-spokes* and *chains*. A *hub-and-spokes* is defined by a larger object surrounded by multiple instances of a smaller object (e.g. a bed between two nightstands; a table surrounded by chairs). A *chain* is defined by a series of objects arranged along a line (e.g. a row of wardrobes against a wall). Figure 5 shows an example of each of these types of superstructures, and Appendix A describes our heuristics for extracting them in detail.

There are other high-level structures we could try to detect that have been exploited in other graphics domains, e.g. grids for inverse procedural model applications [Bokeloh et al. 2010], but such patterns do not occur frequently in our data.

Edge pruning: The graphs as extracted thus far are dense (average of 4 edges per node across all rooms in our dataset), containing many

Room Type	# Nodes	# Nodes (non-wall)	# Edges	# Edges (non-wall)
Bedroom	14.47	10.15	26.43	4.77
Living	14.43	10.12	25.90	4.61
Office	13.93	9.68	25.63	5.41
Bathroom	10.64	6.43	21.36	0.99
Kitchen	15.90	11.53	32.38	8.60

Table 1. Average node and edge count statistics for graphs extracted from SUNCG rooms using the automatic procedure from Section 4. "Non-wall" edges are those where neither endpoint is a wall.

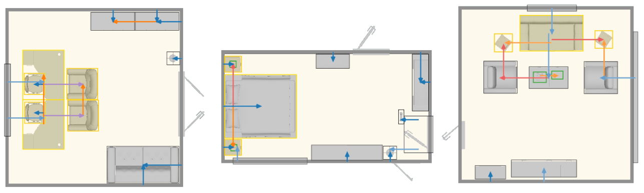


Fig. 6. Examples of relation graphs extracted from training set scenes.

spatially-true but not semantically-meaningful relationships. This density poses two problems for learning a graph generative model. First, very dense graphs will slow down training. Second, and more critically, dense graphs can confuse a neural network model by (1) failing to recognize the most important structural relationships (i.e. it "loses the signal in the noise") or (2) predicting edges which are not self-consistent, i.e. it is impossible to spatially realize the graph. Thus, we prune away 'insignificant' edges from the extracted graphs, keeping only those that reflect the most meaningful relationships. We do this pruning heuristically: some edges are always kept or always deleted based on heuristics about object functionality, and other edges may be kept if they occur frequently enough across the whole dataset of rooms. Appendix A describes our pruning procedure in detail.

Guaranteeing connectivity: Edge pruning may make the graph *disconnected*, i.e. there exists no directed path from the wall nodes to one or more object nodes. As we will describe in Section 6, our scene synthesis process iteratively inserts objects with an inbound edge from an object already in the scene. Since this process can start with only walls in the scene, a scene is not synthesizable if its graph is disconnected. To solve this problem, we find all unreachable nodes and reconnect them to the rest of the graph using a minimum-cost-path-based approach described in Appendix A.

Table 1 shows some statistics for our extracted graphs. Figure 6 shows some example scenes and the graphs extracted from them.

5 GRAPH GENERATION

Our goal is now to learn a generative model from our extracted graphs. Graph generation is a long-studied problem [Erdos and Renyi 1960; Rozenberg 1997]. It is currently experiencing a renaissance driven by deep neural network models. Highly-quality graph generative models have the potential for high impact in computer graphics, as many of the objects graphics researchers study can be naturally represented as graphs: graphic design layouts, urban layouts, curve networks, triangle meshes, etc. To our knowledge, we are the first to apply deep generative graph models to a computer graphics problem. Thus, the goals of this section are twofold. First, we introduce the class of graph generative model we use—autoregressive generation based on message-passing graph convolution—to the graphics community. Second, we describe a specific implementation of this type of generative model with domain-specific design choices for indoor scene relationship graph synthesis.

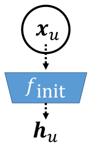
5.1 Autoregressive Graph Generation via Message-Passing Graph Convolution

Our approach to graph generation is based on the framework introduced by [Li et al. 2018b]; for clarity, we highlight the core components of this approach here. The idea behind *autoregressive* graph generation is to construct a graph via a sequence of structure-building decisions, where each decision is computed as a function of the graph that has been built thus far. Specifically, one can construct a graph by iterating the following sequence of decisions:

(1) Should a new node $\mathbf{u}$ be added to the graph? If no, then terminate. If yes:
(2) Should a new edge be connected to $\mathbf{u}$? If no, go to (1). If yes:
(3) Which other node $\mathbf{v}$ in the graph should $\mathbf{u}$ be connected to? Choose one, then go to (2).

On what basis should the generator should make these decisions? Ideally, it would have some way to "look" at the current state of the graph and make a decision based on what it "sees." For image-based generative models, a convolutional neural network (CNN) provides this capability: a CNN can ingest an image and output a distribution over decisions. An image can be interpreted as a graph whose nodes are pixels and whose edges form a regular 2D lattice over these pixels. If we instead use irregularly-structured graphs, is there a way to generalize convolution (and thus CNNs) to operate on such data? The formulation we use is *message passing graph convolution*, which has the following steps:

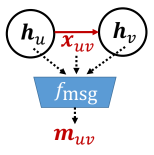
Initialization



Graph properties are represented either as a node feature $\mathbf{x}_u$, describing the properties of the node, or an edge feature $\mathbf{x}_{uv}$, describing relationships between the endpoints of the edge. The graph convolution process uses and updates these properties. For brevity, we consider the version that only updates the node features. To facilitate such update, it is most natural to use a initializer to map the visible node feature x_u into a latent representation $\mathbf{h}_u = f_{\text{init}}(\mathbf{x}_u, \ldots)$, taking the node feature, as well as additional features such as a description of the entire graph, as inputs.

Propagation Similarly to image convolution, graph convolution aggregates information from proximal nodes in the graph. To do so, the following steps are executed:

Compute edge messages: Information propagation in the graph follows edges, which define proximities in the graph. A message function f_{msg} is used to computed the propagated message $\mathbf{m}_{uv}$



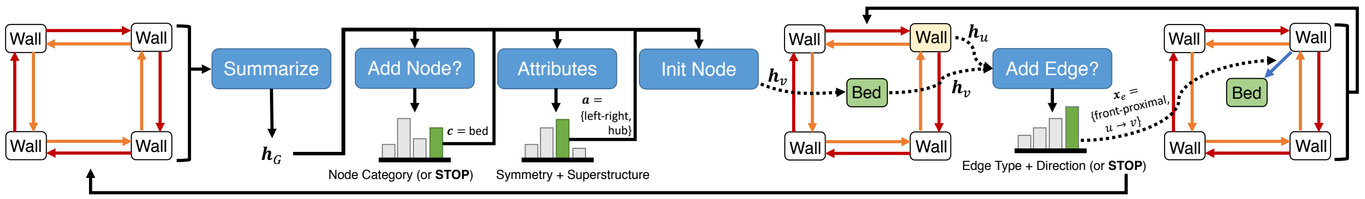
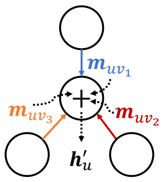


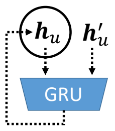
Fig. 7. Overview of the graph generation pipeline. We start from nodes and edges describing the architecture of the room and iteratively add new nodes and edges with a sequence of decision modules that predicts the category of the new node, the symmetry and superstructure type of the new node, and the edges incident to the newly added nodes.

from the edge feature $\mathbf{x}_{uv}$ and the latent node features $\mathbf{h}_u$, $\mathbf{h}_v$. If the edge is directed, the order in which inputs are supplied identifies the direction of the edge.

Aggregate messages. To actually propagate the computed messages, they are gathered at the nodes incident to their associated edges. Since a node can have varying degree, an operation is needed to map the messages to an aggregated message $\mathbf{h}'_u$ with a fixed dimension. This is often done by summing up the messages, though other order-invariant n-ary operations such as mean can also be used. For a directed graph, only the end node of the edge receives the message. Thus, it is often desirable to also have a second function f'_{msg} that computes messages in the reverse direction.

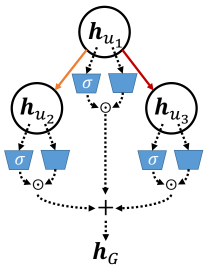


Update node state. Finally, the aggregated message h'_u is used to update the latent node representation h_u. Since we want to keep the latent representation consistent across different steps in the generative model, we use a Gated Recurrent Unit as the update function.



Summarization

In addition to aggregated features at nodes, many tasks require an overall description $\mathbf{h}_G$ of the entire graph. Such a description can be constructed by computing a feature vector for each node, and summing these vectors directly. As different nodes can bear different importance to the entire graph, a gated sum is often used, where a gate function σ is computed and multiplied with each node vector before summing up.



5.2 Generative Model of Scene Relationship Graphs

Our approach to generating scene relationship graphs uses the above building blocks. In particular, we largely follow the design of [Li et al. 2018b], where a series of structural decision modules are used to add new nodes and new edges to the graph in an autoregressive fashion, until completion.

Feature Representation and Initialization Nodes and edges in our graph belong to two major categories. *Architectural* nodes and edges define the room architecture. We always initialize our graph with these, and never predict additional instances. *Object* nodes, and edges connected to them, describe the room layout we want to predict. Table 2 summarizes the input features used for different types of nodes and edges. We use separate f_{init} functions for architectural and object nodes to map their different feature sets to the same latent space. For edge features $\mathbf{x}_{uv}$, information not present (e.g. angle between walls for non-wall edges) is set to 0.

Structure Building Modules Fig 7 shows our pipeline. Starting with architectural nodes and edges, we use several decision modules to iteratively build the graph:

Add Node? At each step, we first predict what node to add. This module performs T rounds of propagation, then applies a neural network f_{add} to predict a discrete distribution over the possible object categories from the graph representation $\mathbf{h}_G$. We also include a "STOP" category which indicates that no more nodes should be added. This module is similar to the image-based category prediction module of prior work [Ritchie et al. 2019]. However, instead of training the module to always pick categories in the same order, we found that a random ordering is sufficient for this task. Random ordering has the benefit of supporting partial graph completion starting from any set of nodes.

Attributes. In addition to the node's category, we also need to know its symmetry type, and whether it belongs to a superstructure. To do so, we apply another neural network f_{sym} to predict a discrete distribution over all possible combinations of symmetry and superstructure types from the same graph representation h_G used in the previous step. To condition on the predicted object category, we use separate network weights for each category.

Add Edge? Finally, we add the new node to the graph and determine the edges incident to it. We use a neural network f_{edge} which takes as input node features $\mathbf{h}_u, \mathbf{h}_v$ for the new node v and existing node u and outputs log probabilities for adding each possible type of edge between those nodes. We compute these log probabilities for all existing nodes u, concatenate them into one distribution, and sample from it. Unlike prior work [Li et al. 2018b], we do not break this step into two modules: *Should Add Edge?* and *Which Edge?*. Instead, we append to the concatenated logits an additional fixed-value logit indicating "STOP." We do so because we find that the predictions of these two modules are often inconsistent: the module can decide to add a new edge but have a high entropy distribution of

Object Type	Included Features
Architectural x_u	Category, Length (walls only)
Object x_u	Category, Symmetry type, Superstructure type
Architectural x_{uv}	Distance, Direction, Angle between walls
Other x_{uv}	Distance, Direction, Support

Table 2. Information our model consumes and/or predicts for different types of nodes and edges in the graph.

possible edges to add. This step is iterated until the network decide that no more edges should be added. Before each iteration, an additional T rounds of propagation is performed. We also use separate network weights for object-architecture edges and object-object edges, since those edges behave differently. Finally, at test time, we reject sampled edges that never occur in the dataset, though this is rare.

Implementation Details We use $T = 3$ rounds of propagation everywhere, and we use a three layer MLP for all structural decision modules. We also use a three layer MLP for f_{msg} during propagation, instead of the single linear layer suggested by [Li et al. 2018b], as this performed significantly better in our experiments. All MLPs used a hidden layer size of 384. For additional robustness, we include the one-hot node representation $\mathbf{x}_u$ in addition to the latent $\mathbf{h}_u$ as input when computing messages, and we include the full graph representation $\mathbf{h}_G$ when computing per-node edge distributions.

Figure 8 shows some graphs generated by this model. They capture important high-level structural patterns and are generally spatially consistent (the next section evaluates this more thoroughly). Occasionally, the model generates inconsistent results. We reject obvious failures cases where the graph is disconnected, acyclic, or contains inconsistent chains. We also address some of the more subtle spatial inconsistencies in the instantiation process, discussed in the following section. Still, most of these issues could be better resolved by adopting a more sophisticated graph generative model, instances of which are rapidly emerging. For example, a model which couples all structure-building decisions through a global latent variable could increase global coherence [Jin et al. 2018]. One could also use reinforcement learning to force the model's output to be spatially realizable [You et al. 2018a].

6 SCENE INSTANTIATION

In this section, we describe our procedure for taking a relationship graph (either generated or manually-authored) and *instantiating* it into an actual 3D scene. Due to the edge pruning steps taken in Section 4, the graphs from which our graph generative model learns (and thus the graphs that it learns to generate) are not complete scene specifications: in general, a graph does not contain enough relationship edges to uniquely determine the spatial positions and orientations of all object nodes. Rather, there is a set of possible object layouts which are consistent with the graph. In other words, the graph defines a set of constraints, and instantiating a layout that satisfies them requires solving a constraint satisfaction problem (CSP). Furthermore, not all scenes within this feasible set are created

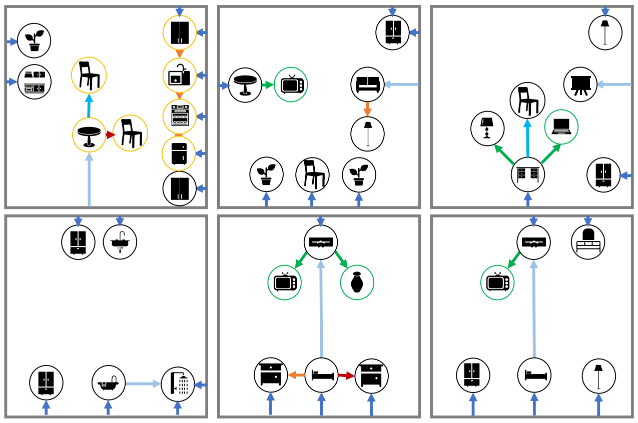


Fig. 8. Scene relationship graphs generated by our model.

equal; some will appear more plausible than others. Specifically, some feasible scenes will respect commonsense layout principles that do not appear in the graph but nevertheless must be followed to ensure plausibility. Thus, we require some kind of prior over scenes to plausibly fill in these gaps left unspecified by the graph. Put another way, we seek not *any* feasible scene, but rather the *most probable* scenes from within the feasible set.

More formally, we require a procedure for sampling from the following conditional probability distribution:

$$p(\mathcal{S}|\mathcal{G}(V,E)) = p(\mathcal{S}) \cdot \prod_{\mathbf{v} \in V} \mathbb{1}(\mathbf{v} \in \mathcal{S}) \cdot \prod_{(\mathbf{u},\mathbf{v},r) \in E} r(\mathcal{S},\mathbf{u},\mathbf{v}) \quad (1)$$

where $\mathcal{S}$ is a scene, $\mathcal{G}(V,E)$ is a graph with vertices V and edges E, and $r(\cdot)$ is a predicate function indicating whether the relationship implied by the edge $(\mathbf{u},\mathbf{v},r)$ is satisfied in $\mathcal{S}$. The conditional probability factors as the product of the *prior probability* over scenes $p(\mathcal{S})$ (which is implied by the dataset) and a product of $\{0,1\}$ constraint indicator functions. With multiple constraints, much of the work involved in sampling this distribution is in finding a scene $\mathcal{S}$ with nonzero probability. For this, we adopt a backtracking search strategy, as is common for CSP solving: we instantiate objects one at a time until some relationship constraint is violated, at which point we roll back one or more steps and retry. Within this framework, we also require (a) a means of selecting values for an object's spatial configuration variables (position, orientation, size), and (b) a way to evaluate $p(\mathcal{S})$. Here, we kill two birds with one stone: we generate configuration values via neural nets which are trained to sample from an approximation of the conditional probability in Equation 1. This makes our instantiation algorithm a *neurally-guided search* procedure [Ritchie et al. 2016; Vijayakumar et al. 2018].

The rest of this section explains the design decisions behind our search procedure: the order in which to instantiate, the neural nets used to sample object configurations, and our backtracking strategy.

6.1 Object Instantiation Order

The order in which a CSP solver assigns values to variables has significant impact on its performance. A common ordering strategy is the Most Constrained Variable (MCV) heuristic: assign values to variables that participate in the most constraints first [Russell and Norvig 2009]. The intuition here is that such assignments will cause the search to "fail fast," allowing it to correct an infeasible assignment to one or more of those variables without requiring significant backtracking.

Our algorithm for determining the order in which to instantiate objects follows a similar logic. It uses the structure of the graph, along with statistics about typical sizes for nodes of different categories, to determine an ordering that leads to fast failure and minimizes backtracking. Most of these principles are not specific to the indoor scene domain and would be valid for many constraint-based spatial layout problems with constraints derived from a graph.

Ordering preliminaries: We require that our ordering algorithm sort scene objects topologically: an object cannot be instantiated until all its inbound neighbors which constrain its placement have also been instantiated. Among the different possible topological sorts, we additionally require that ours follows a depth-first ordering: when a node is added to the scene, all of its descendants must be added before any of its non-descendants. This requirement leads to insertion orders that "grow" inward from the walls, which is more likely to instantiate coherent sub-parts of the scene together (and which we exploit during backtracking.

Constraint-based ordering: For most graphs, there exist many possible orderings that satisfy the above requirements. Thus, we additionally sort nodes based on *how constrained* they are. This measure is based on the number of inbound edges to a node, as an object with a specified location relative to multiple other objects has fewer possible valid placements. Specifically, we define the score $C_{\rightarrow}(\mathbf{v})$ to be the weighted sum of node $\mathbf{v}$'s inbound edges, where adjacent, proximal, and distant edges are weighted in a 3:2:1 ratio. For nodes with the same $C_{\rightarrow}$ score, we break ties based on which node would be *most constraining* if it were to be instantiated next. We define a second score $C_{\leftarrow}(\mathbf{v})$ as:

$$C_{\leftarrow}(\mathbf{v}) = \sum_{\mathbf{u} \in D_{\mathcal{G}}[\mathbf{v}]} C_{\rightarrow}(\mathbf{u}) \cdot \mathbb{E}[\text{Size}(\mathbf{u})]$$

where $D_{\mathcal{G}}[\mathbf{v}]$ are the (inclusive) descendants of $\mathbf{v}$ and the expected value of the size of a node is the average 2D projected bounding box area of objects of that node's category in the dataset. In other words, a node is very constraining if instantiating it would lead to the insertion of many large objects which are themselves highly constrained (i.e. have relatively fixed positions).

Domain-specific ordering principles: We use a few ordering principles which are specific to indoor scenes. First, we impose additional requirements on the object insertion ordering to preserve the integrity of superstructures. When a hub node is added to the order, we add its spoke nodes before any other outbound neighbors. When a chain start node is added, we add all the nodes in the chain, followed by the union of their descendants. In addition, we place all second-tier (i.e. supported) objects after all first-tier objects.

We note that prior scene synthesis methods which iteratively construct scenes object-by-object have also had to contend with the object-ordering problem. At least one prior work has used a random ordering of objects [Wang et al. 2018]. Another possibility is to use an ad-hoc "importance" order, i.e. ordering objects by a combination of their size and frequency in the dataset [Ritchie et al. 2019]. The latter strategy can be seen as a form of Most Constrain(ing) Variable heuristic. Our graph-based representation provides a richer set of information from which to derive a more sophisticated ordering.

6.2 Neurally-Guided Object Instantiation

Once we have selected an object of category c to add to the scene, as prescribed by the ordering above, we must propose a *configuration* for it: its location x, orientation θ, and physical dimensions $\mathbf{d}_{xy}$. Ideally, we seek a generator function $g(\mathbf{x}, \theta, \mathbf{d}_{xy} | c, \mathcal{S}, \mathcal{G}(V, E))$ which outputs values with probability proportional to the true conditional scene probability $p(\hat{\mathcal{S}} = \mathcal{S} \cup \{(c, \mathbf{x}, \theta, \mathbf{d}_{xy})\} | \mathcal{G}(V, E))$ in Equation 1. In other words, we need to design an importance sampler for p.

As in prior work on scene synthesis by iterative object insertion [Ritchie et al. 2019], we decompose the configuration generator as $g(\mathbf{x}, \theta, \mathbf{d}_{xy} | \cdot) = g(\mathbf{d}_{xy} | \theta, \mathbf{x}, \cdot) g(\theta | \mathbf{x}, \cdot) g(\mathbf{x} | \cdot)$ according to the chain rule, i.e. we sample location, then orientation, then dimensions.

Location Since p factorizes as the product of the scene prior probability $p(\mathcal{S})$ and the constraint functions, one possible strategy is to use a learned prior over object locations in scenes for $g(\mathbf{x}|\cdot)$. For our prior, we adopt the location prediction module from [Ritchie et al. 2019], which is a state-of-the-art image-based prior over scenes. It is a fully-convolutional network (FCN) that takes as input a top-down view of the scene $\mathcal{S}$ and produces an image-space probability distribution over possible locations for the next object to insert. While this is a good importance sampler for $p(\mathcal{S})$, it is far from ideal when used to sample the conditional distribution in Equation 1, as many proposed locations will violate the constraints (Table 3).

We would prefer our importance sampler $g(\mathbf{x}|\cdot)$ to respect constraints by construction, rather than by rejection. To do this, we modify the FCN architecture of [Ritchie et al. 2019] to be aware of the relationship graph. Figure 9 shows a schematic of this architecture. Rather than predicting a location distribution for a target object in the absolute coordinate frame of a scene image, it predicts a distribution in a *local* coordinate frame relative to one of that object's inbound neighbors in the graph; we call these neighbors *anchors*. Furthermore, the FCN takes an additional input in the form of the type of relationship edge between the anchor and the target (e.g. *left, adjacent*), turning it into a conditional FCN (CFCN). This additional input is injected into the network via Featurewise Linear Modulation (FiLM), a process that applies a learned scale and shift to the output of every convolutional layer [Perez et al. 2018]. If there is more than one anchor object, the module predicts a relative location distribution for each of them, transforms these into the global coordinate frame, and combines them via multiplication and renormalization (Figure 10). This local location prediction strategy can be seen as a form of visual attention [Xu et al. 2015], supervised by the structure of the graph. Conditioning the network to be graph-aware in this way significantly reduces the percentage of proposed locations which violate constraints (Table 3, Figure 11).

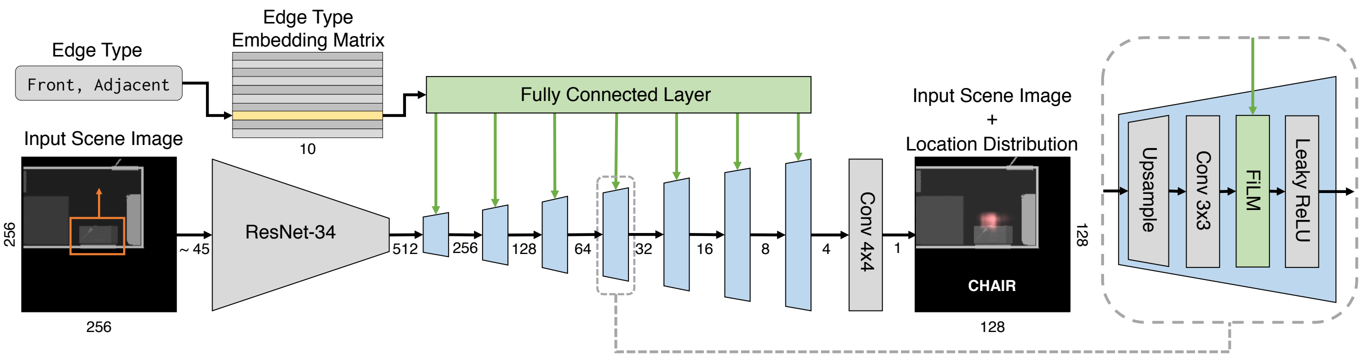


Fig. 9. Architecture of our graph-conditional location prediction network. We use a fully-convolution network (FCN) architecture to predict a 2D distribution of possible object locations, in the relative coordinate frame of an anchor object (orange region in the input image). The network's output is conditioned on a type of relationship edge via featurewise linear modulation (FiLM) [Perez et al. 2018]. The network simultaneously predicts distributions for all categories; here we visualize the slice for "chair."

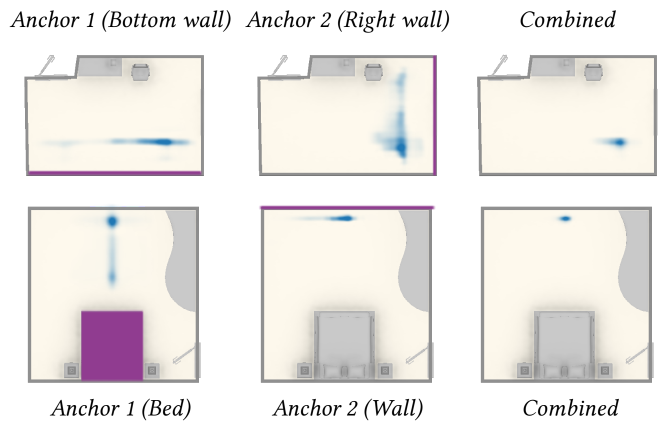


Fig. 10. Combining multiple anchor-relative location distributions into one global distribution. The anchor object is highlighted in purple. Note how the product of the two anchor-relative distributions leaves an unambiguous signal that the bed should be in the corner *(top row)* and that the TV stand should be directly across from the bed *(bottom row)*.

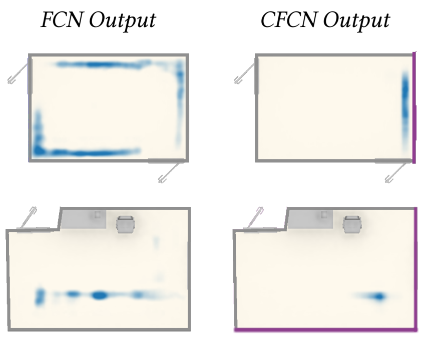


Fig. 11. Output of the previous unconditional fully convolution network (FCN) compared with our edge-conditional CFCN. The FCN predicts mostly plausible locations, but is oblivious to the structure of the graph and thus likely to suggest constraint-violating locations. Once again, anchor objects (e.g. edge start nodes) ar highlighted in purple in the righthand column.

We create training data for this network by randomly removing a subset of objects from a scene, choosing an object with at least one outgoing edge to one of the removed objects as the anchor, and tasking it with predicting the location of that object. We also perform data augmentation by exploiting the symmetry type of the anchor object. For example, if the anchor has a left/right reflectional symmetry, we randomly choose whether to reflect the input image across the anchor's symmetry plane. More generally, a symmetry type implies the object has some number of equivalent coordinate frames, and we pick one of them at random during training.

Orientation and dimensions To propose orientations and physical dimensions for objects, we also adopt the image-based modules from previous work [Ritchie et al. 2019]. These are conditional variational autoencoders (CVAEs) which take a top-down scene image as input (centered on the object in question) and output either a front-facing vector or 2D projected bounding box dimensions for the object. One could imagine modifying these networks to make them graph-aware, in a similar manner to the location module described above. However, we found this extra complexity not to be necessary, as the object's location and spatial context tend to provide sufficient conditioning information to make reasonable choices.

Backtracking: Even with the neurally-guided location sampling procedures described above, instantiating objects in the order prescribed by Section 6.1 can still lead to scenes which do not satisfy all the relationships given by the graph. This becomes more likely as the number of graph edges (and hence the number of constraints) in the graph increases. Thus, we use a backtracking search procedure to roll back previously-instantiated objects when faced with a constraint that cannot be satisfied. Appendix B provides more details on our backtracking policy. As part of this policy, we also allow the search process to begin violating constraints as it accumulates a large number of backtracking steps. This is sometimes necessary to instantiate a graph at all, as our graph generative model does

Method	Rejections	Backtracks	Violations	Time
Random Location Sample	0.283	13.017	0.739	45.167
FCN-based Location sample	0.050	9.317	0.453	19.367
CFCN + All Heuristics	0.067	3.767	0.057	10.783
No Pruning	0.067	19.067	1.704	46.429
Half Pruning	0.083	10.283	0.475	20.889
No Constraint Based Ordering	0.117	6.133	0.153	17.505

Table 3. Evaluating the performance of different location sampling methods (*Top*) and different graph generation/instantiation strategies (*Bottom*) in terms of the average number of uninstantiable graphs *(Rejections)*, the average number of backtracking steps initiated per object insertion *(Backtracks)*, the average number of graph constraints violated in the final output scene *(Violations)*, and the average time taken in seconds to instantiate a scene *(Time)*.

not (and in general cannot) guarantee that the graphs it outputs are physically realizable. Appendix B also describes our policy for this form of *constraint relaxation*, which quantifies the extent to which an object is in violation of its constraints and allows this value to gradually increase as a function of the number of times that attempting to instantiate the object has led to backtracking.

Table 3 shows some statistics for our backtracking search procedure on bedrooms. We report the frequency of rejected insertions, backtracking steps, and constraint violations in final graphs. To provide baselines, we perform an ablation study in which we use search with no neural guidance (i.e. random sampling of object configurations), the non-conditional FCN which is oblivious to the graph, and using our edge-conditional CFCN. Performance on all metrics improves with more specific neural guidance. Similarly, our edge pruning and constraint-based ordering heuristics have significant impact on backtracking performance. Removing them, or using a pruning threshold that is half as strict, leads to worse performance.

7 RESULTS & EVALUATION

In this section, we present qualitative and quantitative results demonstrating the utility of the PlanIT approach to scene synthesis and comparing it to prior scene synthesis methods.

Synthesizing new scenes: Figure 1 shows scenes synthesized by PlanIT, along with their corresponding graphs. The graph representation contains information that allows the instantiation phase to realize otherwise challenging layouts, such as the chains of wardrobes and kitchen cabinets, or the office chairs tucked between desks and walls. As mentioned earlier in Section 6, a relationship graph usually does not uniquely specify a scene. To illustrate this visually, Figure 12 shows examples of instantiating the same graph multiple times. We also evaluate the generalization behavior of our model by computing the average similarity of a generated scene to its most similar scene in the training set [Ritchie et al. 2019]. The average similarity is 0.781 ± 0.049. For reference, the average similarity of a training set scene to the most similar other scene in the training set is 0.804 ± 0.061). This indicates that our model does not simply memorize, and that it generates scenes with at least as much internal diversity as the training scenes.

Partial graph completion: Because it instantiates scenes object-by-object, PlanIT supports completion of partial scenes, as does prior work [Ritchie et al. 2019]. One unique property of PlanIT, however, is that it also supports synthesis from a partial *graph*. The ability to synthesize a scene from a high-level partial specification of what it should contain can be useful, for example in dialogue-based interfaces (e.g. when designing a bedroom for twins: "show me bedrooms with two single beds and a nightstand between them"). Figure 13 shows some examples of synthesis by partial graph completion. This is similar to partial scene completion, but the input need not specify the geometry or even the placement of the initial objects. Prior object-centric scene synthesis methods either do not support partial structure completion [Li et al. 2018a] or must invoke a more expensive optimization process to do so [Zhang et al. 2018].

Perceptual Study: As our first quantitative evaluation of scenes generated by PlanIT, we conducted a two-alternative forced choice (2AFC) perceptual study on Amazon Mechanical Turk comparing images of its scenes to those generated by other methods. Participants were shown two top-down scene images side by side and asked to pick the more plausible one. These images were rendered using solid colors for each object category, to factor out effects of material appearance. For each comparison and each room type, we recruited 10 participants. Each participant performed 55 comparisons; 5 of these were "vigilance tests" comparing against a randomly jumbled scene (i.e. the random scene should always be dis-preferred). We filtered out participants who did not pass all vigilance tests.

Table 4 shows the results of this experiment. PlanIT consistently outperforms GRAINS, the previous state-of-the-art object-centric scene synthesis system. When compared to the Fast & Flexible image-based method, PlanIT is comparable across most scenes (i.e. differences are not statistically significant). With the exception of living rooms, PlanIT's output scenes do not fare as well when compared to scenes created by people. These results suggest that the PlanIT can deliver comparable output quality to other state-of-the-art methods, while also supporting new applications and usage modes. However, the constraint satisfaction problems imposed by generated relationship graphs are still difficult to solve, and thus PlanIT is not yet at the level of human-like scene design capabilities.

Real vs. Synthetic Classification Experiment: In addition to asking people for their preferences, we also ask machines: we train a classifier to distinguish between "real" scenes (from the training set) and "synthetic" scenes (generated by a learned model). We adopt the same experiment settings as prior work [Ritchie et al. 2019]: we use a Resnet34 that takes as input the same top-down scene representation used by the location suggestion FCN from Section 6.2, and render all scenes into the same representation before feeding them to the classifier. The classifier is trained with 1600 scenes, half from the training set and half generated. We evaluate accuracy on 320 held out test scenes.

Table 5 shows the results of this experiment. Our method performs similarly to the previous image based methods. It is not surprising PlanIT is not in first place on this metric. After all, the graphs

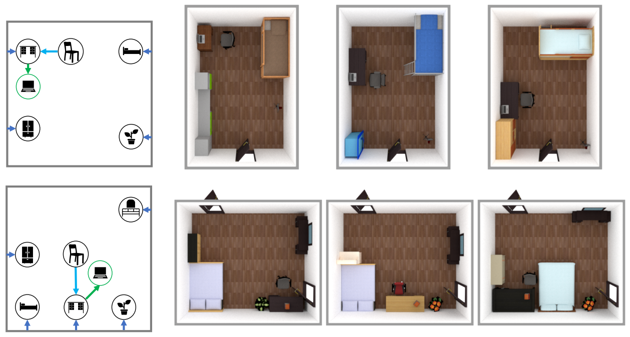


Fig. 12. A graph does not uniquely describe a scene; here we show multiple instantiations of the same graph.

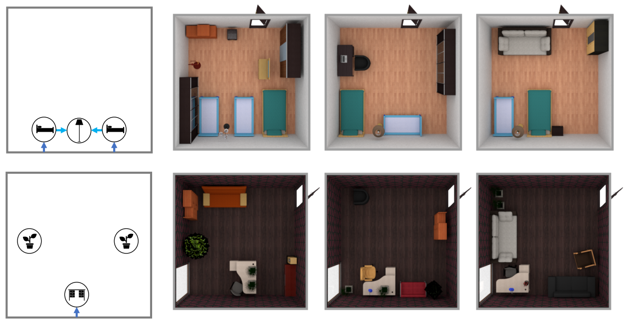


Fig. 13. Completed scenes from a partial graph manually constructed from a natural language description. Top: two single beds by the bottom wall, with a floor lamp in between; Bottom: An office with a desk near a wall and some plants. Our model is able to synthesize a variety of scenes that adhere to the description.

	Ours vs.		
Room Type	GRAINS	Fast & Flexible	SUNCG
Bedroom	**59.3 ± 4.3**	48.2 ± 4.4	44.1 ± 4.8
Living	**82.9 ± 3.4**	45.9 ± 6.6	44.9 ± 5.2
Office	**76.3 ± 5.6**	**57.4 ± 5.1**	41.3 ± 6.6
Bathroom	—	42.8 ± 4.4	34.0 ± 4.0
Kitchen	—	51.2 ± 5.1	29.9 ± 4.2

Table 4. Percentage (± standard error) of forced-choice comparisons in which scenes generated by our method are judged as more plausible than scenes from another source. Higher is better. Bold indicate our scenes are preferred with > 95% confidence; gray indicates our scenes are dis-preferred with > 95% confidence; regular text indicates no preference. — indicates unavailable results.

Method	**Accuracy**
Deep Priors [2018]	84.69
Fast & Flexible [2019]	58.75
Ours	63.13

Table 5. Real vs. synthetic classification accuracy for scenes generated by different methods. Lower (closer to 50%) is better. Adapted from [Ritchie et al. 2019], Table 2.

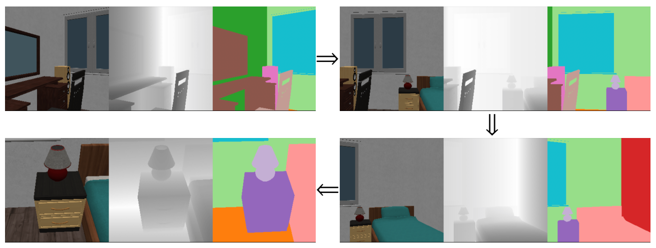


Fig. 14. Our approach can be used to generate specific, task-relevant scenes for use in 3D simulation. In this example, we generate a bedroom with a nightstand and lamp. Then we use the MINOS [Savva et al. 2017] simulator to extract frames for color, depth and semantic segmentation during a navigation trajectory where the goal is to locate the lamp on the nightstand (sequence is clockwise from top left to bottom left).

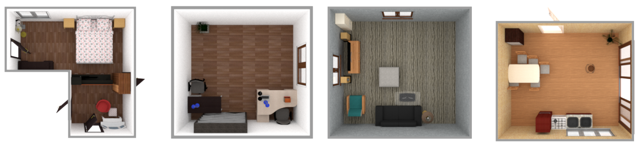


Fig. 15. Typical failure cases for our model. From left to right: TV stand blocks access to part of the room; loudspeakers placed behind TV stand instead of beside it; this particular desk cannot be accessed when used in a corner; failure to precisely arrange dining chairs around the table.

it generates are based on training graphs which pruned out many edges. Essentially, we have discarded some noise which is in the data, and the classifier is likely picking up on the fact that this noise is missing.

Timing: We train and evaluate our model on a 12-core Intel i7-6850K machine with 32GB RAM and an NVIDIA GTX 1080Ti GPU. The graph generative model is trained on the CPU; this takes 15 hours for bedrooms, kitchens and toilets, and 10 hours for living rooms and offices. The other modules are trained on the GPU, following suggestions by [Ritchie et al. 2019].

At test time, it takes ≈ 0.2 seconds to sample a graph. It takes on average 10 seconds to instantiate a graph, with graphs requiring minimal backtracking taking ≈ 1.5*s* and graphs with repeated backtracking taking up to 1 minute.

Generating Custom Virtual Agent Training Environments: Partial graphs can be used to specify objects that must be present in a scene, thus allowing us to generate custom scenes that can be useful when specific tasks need to be performed in 3D simulation. Figure 14 shows an example.

8 CONCLUSION

In this paper, we presented PlanIT, a new conceptual framework for layout generation which decomposes the problem into two stages: planning and instantiation. We demonstrated a concrete implementation of the PlanIT framework for the domain of indoor scene synthesis. Our method *plans* a scene by generating an object relation graph, and then it *instantiates* that scene by sampling compatible object configurations. To provide training data for our system, we described a heuristic approach for extracting relationship graphs from unstructured scenes. We then described a generative model for such graphs, based on deep graph convolutional networks. Finally, we showed how to instantiate a relationship graph by searching for object configurations that satisfy the graph's constraints, using image-based convolutional networks. PlanIT's two-stage synthesis approach leads to more modeling flexibility and applicability to more use cases, and our experimental results show that this adding flexibility does not come at the cost of decreased output quality.

Our method is not without its limitations. Figure 15 shows some failure cases of our model. These arise from the instantiation module not modeling the functionality of objects and spaces (left) and making placement errors that deviate from a strict expected arrangement template. These could be addressed by adding "empty space" as a first-class entity in the model and by refining our treatment of superstructures to guarantee more precise arrangements.

Our results also depend critically on the quality of the training graphs extracted from the input scene dataset. Our process for determining which edges to include in the graphs is heuristic. While we believe our design choices are justified, our graph extraction step could very well filter out important relationships or include spurious ones. What *are* the most salient relationships in a scene? This question has different answers, depending on one's interpretation. Do we care about what is most salient to a machine trying to instantiate the graph? Or what is most salient to a person trying to specify a scene via a graph? Is it possible to *learn* how to extract good relationship graphs, perhaps by using reinforcement learning to extract graphs which lead to graph generative models with high performance on some metric? Further research is needed here.

Our graph representation itself is also limited by the small set of relationships it encodes. Support relationships and frequent spatial relationships only scratch the surface of what is possible in terms of analyzing the functionality of 3D objects and scenes [Hu et al. 2018]. Fortunately, as our graph generative model architecture is quite general, our relationship graph format is quite extensible. It would be interesting to explore augmenting it with more nuanced functional relationships (e.g. containment, affordances for human activity) to help constrain scene synthesis toward producing more usable and interactive interior spaces.

Dividing the scene synthesis problem into two phases, with an intermediate graph representation, makes PlanIT flexible and applicable to more use cases. But it also turns scene instantiation into a constraint satisfaction problem, which takes time to solve. If we give up the ability to synthesize scenes from complete or partial graphs, and instead focus on synthesizing scenes from scratch, could we combine the graph generation and scene instantiation phases? That is, could one design a tightly-integrated generative model that generates a graph while instantiating a scene from it in lock-step, where each representation provides feedback to the other? This seems like a fruitful direction for future work.

There are also many opportunities to extend and apply a system like PlanIT. For example, it would be valuable to develop a natural language interface for constructing partial input graphs, as alluded to earlier. What type of language must such a system support to be useful? What graph representation maps most naturally to this language? There exists prior work in language-based scene creation [Chang et al. 2015, 2014], including recent work that uses a graph-based intermediate representation [Ma et al. 2018a]. However, it constructs scenes by retrieving parts of scenes from a database; new possibilities are opened up by a system that can synthesize truly new scenes from a partial graph.

Last but not least, it is important to explore the applicability of the PlanIT framework to other layout generation applications. Layout is a critical subproblem in many graphics and design domains. Could one build a graph-based plan-and-instantiate framework for producing constrained web designs, or other types of graphic designs? The hybrid nature of our framework could be very useful: the graph could dictate the functional layout of design elements, while the image-based priors could focus on aesthetic concerns.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful suggestions. Scene renderings shown in this paper were created using the Mitsuba physically-based renderer [Jakob 2010]. This work was supported in part by NSF award #1753684, a hardware donation from Nvidia, and with the support of the Technical University of Munich-Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement no 291763.

REFERENCES

Peter Anderson, Basura Fernando, Mark Johnson, and Stephen Gould. 2016. Spice: Semantic propositional image caption evaluation. In *European Conference on Computer Vision (ECCV)*. Springer, 382–398.

Martin Bokeloh, Michael Wand, and Hanspeter Seidel. 2010. A connection between partial symmetry and inverse procedural modeling. In *SIGGRAPH 2010.*

Angel Chang, Will Monroe, Manolis Savva, Christopher Potts, and Christopher D. Manning. 2015. Text to 3D Scene Generation with Rich Lexical Grounding. In *ACL 2015.*

Angel X Chang, Manolis Savva, and Christopher D Manning. 2014. Learning Spatial Knowledge for Text to 3D Scene Generation. In *Empirical Methods in Natural Language Processing (EMNLP).*

Chaos Group. 2018. Putting the CGI in IKEA: How V-Ray Helps Visualize Perfect Homes. https://www.chaosgroup.com/blog/putting-the-cgi-in-ikea-how-v-ray-helps-visualize-perfect-homes. Accessed: 2018-10-13.

Kang Chen, Yukun Lai, Yu-Xin Wu, Ralph Robert Martin, and Shi-Min Hu. 2014. Automatic semantic modeling of indoor scenes from low-quality RGB-D data using contextual information. *ACM Transactions on Graphics* 33, 6 (2014).

Angela Dai, Daniel Ritchie, Martin Bokeloh, Scott Reed, Jürgen Sturm, and Matthias Nießner. 2018. ScanComplete: Large-Scale Scene Completion and Semantic Segmentation for 3D Scans. In *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE.*

Abhishek Das, Samyak Datta, Georgia Gkioxari, Stefan Lee, Devi Parikh, and Dhruv Batra. 2018. Embodied Question Answering. In *CVPR.*

P. Erdos and A Renyi. 1960. On the Evolution of Random Graphs. In *PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES.* 17–61.

Matthew Fisher, Daniel Ritchie, Manolis Savva, Thomas Funkhouser, and Pat Hanrahan. 2012. Example-based Synthesis of 3D Object Arrangements. In *SIGGRAPH Asia 2012.*

Matthew Fisher, Manolis Savva, Yangyan Li, Pat Hanrahan, and Matthias Nießner. 2015. Activity-centric Scene Synthesis for Functional 3D Scene Modeling. (2015).

Qiang Fu, Xiaowu Chen, Xiaotian Wang, Sijia Wen, Bin Zhou, and Hongbo Fu. 2017. Adaptive Synthesis of Indoor Scenes via Activity-associated Object Relation Graphs. In *SIGGRAPH Asia 2017.*

Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. *CoRR* arXiv:1704.01212 (2017).

Daniel Gordon, Aniruddha Kembhavi, Mohammad Rastegari, Joseph Redmon, Dieter Fox, and Ali Farhadi. 2018. IQA: Visual Question Answering in Interactive Environments. In *CVPR.*

R. Hu, M. Savva, and O. van Kaick. 2018. Functionality Representations and Applications for Shape Analysis. *Computer Graphics Forum* 37, 2 (2018), 603–624.

Wenzel Jakob. 2010. Mitsuba renderer. http://www.mitsuba-renderer.org.

Wengong Jin, Regina Barzilay, and Tommi S. Jaakkola. 2018. Junction Tree Variational Autoencoder for Molecular Graph Generation. In *ICML 2018*.

Justin Johnson, Agrim Gupta, and Li Fei-Fei. 2018. Image generation from scene graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*.

Justin Johnson, Ranjay Krishna, Michael Stark, Li-Jia Li, David Shamma, Michael Bernstein, and Li Fei-Fei. 2015. Image retrieval using scene graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*.

Z. Sadeghipour Kermani, Z. Liao, P. Tan, and H. Zhang. 2016. Learning 3D Scene Synthesis from Annotated RGB-D Images. In *Eurographics Symposium on Geometry Processing*.

Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *CoRR abs/ 1609.02907* (2016).

Ranjay Krishna, Yuke Zhu, Oliver Groth, Justin Johnson, Kenji Hata, Joshua Kravitz, Stephanie Chen, Yannis Kalantidis, Li-Jia Li, David A Shamma, et al. 2017. Visual genome: Connecting language and vision using crowdsourced dense image annotations. *International Journal of Computer Vision* 123, 1 (2017), 32–73.

Danny Lange. 2018. Unity and DeepMind partner to advance AI research. https://blogs.unity3d.com/2018/09/26/unity-and-deepmind-partner-to-advance-ai-research. Accessed: 2018-10-13.

Manyi Li, Akshay Gadi Patil, Kai Xu, Siddhartha Chaudhuri, Owais Khan, Ariel Shamir, Changhe Tu, Baoquan Chen, Daniel Cohen-Or, and Hao Zhang. 2018a. GRAINS: Generative Recursive Autoencoders for INdoor Scenes. *CoRR* arXiv:1807.09193 (2018).

Yikang Li, Wanli Ouyang, Bolei Zhou, Kun Wang, and Xiaogang Wang. 2017. Scene graph generation from objects, phrases and region captions. In *ICCV*.

Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. 2018b. Learning deep generative models of graphs. *CoRR abs/1803.03324* (2018).

Tianqiang Liu, Siddhartha Chaudhuri, Vladimir G. Kim, Qi-Xing Huang, Niloy J. Mitra, and Thomas Funkhouser. 2014. Creating Consistent Scene Graphs Using a Probabilistic Grammar. In *SIGGRAPH Asia 2014*.

Cewu Lu, Ranjay Krishna, Michael Bernstein, and Li Fei-Fei. 2016. Visual relationship detection with language priors. In *European Conference on Computer Vision (ECCV)*. Springer, 852–869.

Rui Ma, Akshay Gadi Patil, Matthew Fisher, Manyi Li, SÃűren Pirk, Binh-Son Hua, Sai-Kit Yeung, Xin Tong, Leonidas Guibas, and Hao Zhang. 2018a. Language-driven synthesis of 3D scenes from scene databases. In *SIGGRAPH Asia 2018*.

Rui Ma, Akshay Gadi Patil, Matt Fisher, Manyi Li, Soren Pirk, Binh-Son Hua, Sai-Kit Yeung, Xin Tong, Leonidas J. Guibas, and Hao Zhang. 2018b. Language-Driven Synthesis of 3D Scenes Using Scene Databases. *ACM Transactions on Graphics* 37, 6 (2018).

Paul Merrell, Eric Schkufza, Zeyang Li, Maneesh Agrawala, and Vladlen Koltun. 2011. Interactive Furniture Layout Using Interior Design Guidelines. In *SIGGRAPH 2011*.

Vittorio Ferrari Paul Henderson, Kartic Subr. 2018. Automatic Generation of Constrained Furniture Layouts. *CoRR* arXiv:1711.10939 (2018).

Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron C. Courville. 2018. FiLM: Visual Reasoning with a General Conditioning Layer. In *AAAI 2018*.

Planner5d. 2017. Home Design Software and Interior Design Tool ONLINE for home and floor plans in 2D and 3D. https://planner5d.com. Accessed: 2017-10-20.

Siyuan Qi, Yixin Zhu, Siyuan Huang, Chenfanfu Jiang, and Song-Chun Zhu. 2018. Human-centric Indoor Scene Synthesis Using Stochastic Grammar. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

Daniel Ritchie, Anna Thomas, Pat Hanrahan, and Noah D. Goodman. 2016. Neurally-Guided Procedural Models: Amortized Inference for Procedural Graphics Programs using Neural Networks. In *NIPS 2016*.

Daniel Ritchie, Kai Wang, and Yu an Lin. 2019. Fast and Flexible Indoor Scene Synthesis via Deep Convolutional Generative Models. In *CVPR 2019*.

RoomSketcher. 2017. Visualizing Homes. http://www.roomsketcher.com. Accessed: 2017-11-06.

Grzegorz Rozenberg (Ed.). 1997. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA.

Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.

Manolis Savva, Angel X. Chang, Alexey Dosovitskiy, Thomas Funkhouser, and Vladlen Koltun. 2017. MINOS: Multimodal Indoor Simulator for Navigation in Complex Environments. *arXiv:1712.03931* (2017).

Shuran Song, Fisher Yu, Andy Zeng, Angel X Chang, Manolis Savva, and Thomas Funkhouser. 2017. Semantic Scene Completion from a Single Depth Image. *CVPR 2017*.

Ashwin J. Vijayakumar, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. In *ICLR 2018*.

Kai Wang, Manolis Savva, Angel X. Chang, and Daniel Ritchie. 2018. Deep Convolutional Priors for Indoor Scene Synthesis. In *SIGGRAPH 2018*.

Danfei Xu, Yuke Zhu, Christopher B Choy, and Li Fei-Fei. 2017. Scene graph generation by iterative message passing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Vol. 2.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. 2015. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. *CoRR* arXiv:1502.03044 (2015).

Kun Xu, Kang Chen, Hongbo Fu, Wei-Lun Sun, and Shi-Min Hu. 2013. Sketch2Scene: Sketch-based Co-retrieval and Co-placement of 3D Models. In *SIGGRAPH 2013*.

Ken Xu, James Stewart, and Eugene Fiume. 2002. Constraint-based automatic placement for scene composition. In *Graphics Interface*, Vol. 2. 25–34.

Sijie Yan, Yuanjun Xiong, and Dahua Lin. 2018. Spatial Temporal Graph Convolutional Networks for Skeleton-Based Action Recognition. In *AAAI*.

Jianwei Yang, Jiasen Lu, Stefan Lee, Dhruv Batra, and Devi Parikh. 2018. Graph r-cnn for scene graph generation. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 670–685.

Jiaxuan You, Bowen Liu, Rex Ying, Vijay S. Pande, and Jure Leskovec. 2018a. Graph Convolutional Policy Network for Goal-Directed Molecular Graph Generation. In *NeurIPS 2018*.

Jiaxuan You, Rex Ying, Xiang Ren, William L. Hamilton, and Jure Leskovec. 2018b. GraphRNN: A Deep Generative Model for Graphs. In *ICML 2018*.

Lap-Fai Yu, Sai-Kit Yeung, Chi-Keung Tang, Demetri Terzopoulos, Tony F. Chan, and Stanley J. Osher. 2011. Make It Home: Automatic Optimization of Furniture Arrangement. In *SIGGRAPH 2011*.

Rowan Zellers, Mark Yatskar, Sam Thomson, and Yejin Choi. 2018. Neural Motifs: Scene Graph Parsing with Global Context. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5831–5840.

Zaiwei Zhang, Zhenpei Yang, Chongyang Ma, Linjie Luo, Alexander Huth, Etienne Vouga, and Qixing Huang. 2018. Deep Generative Modeling for Scene Synthesis via Hybrid Representations. *CoRR* arXiv:1808.02084 (2018).

A GRAPH EXTRACTION HEURISTICS

This appendix provides more detail about the automatic heuristics we use for extracting relationship graphs from 3D scenes. The complete source code for this procedure is also available online at https://github.com/brownvc/planit.

A.1 Detecting "Superstructures"

The rules for detecting and extracting superstructures are:

Hub-and-spokes: We detect hubs by searching for graph nodes with multiple outbound edges to nodes representing smaller objects of the same category. Such a node is a hub if the arrangement of those neighbor nodes (spokes) is invariant under the node's symmetry group (e.g. a spoke to the left and a spoke to the right, for a node with left-right reflectional symmetry). The hub node inherits all the inbound edges of its spokes. The spokes in turn discard any inbound edges except for the edge from the hub and any adjacent relationships. The reasoning here is that the hub is the primary cue for the placement of the spokes; the only other relevant information to maintain is whether the spokes are directly adjacent to any other objects.

Chain: We detect two variants of 'chain" structures. First, we detect any sequence of three or more object nodes of a similar size connected by *adjacent* edges in the same direction. This condition captures functional arrangements where a larger structure is assembled from multiple contiguous parts (e.g. kitchen cabinetry). We also detect any sequence of three or more instances of the same object connected by *proximal* edges in the same direction. This condition captures commonly-occurring structures with more aesthetic purpose (e.g. a row of plants). Since the spatial configuration of the chain is completely determined by its start and end nodes, chain intermediate nodes discard all of their inbound edges and inherit whatever inbound edges are common to both the start and end node.

A.2 Edge Pruning

The rules for pruning the initial set of extracted graph edges are:

Kept edges: We always retain support edges, wall $\longrightarrow$ object *adjacent* edges, and superstructure edges, as these are critically important for determining object placement. In an effort to retain edges that reflect layout intentionality, we also retain all *adjacent* edges between objects of the same category, as well *proximal* edges originating from an object with a unique front direction (i.e. not symmetric or left-right reflectionally symmetric). These latter edges correspond to an object "pointing at" something nearby (e.g. arm chair pointing at a television).

Deleted edges: We always delete wall $\longrightarrow$ object edges that are not *adjacent*, to reduce the overwhelmingly large number of such edges (44 – 59% of all '$\longrightarrow$ object' edges in the initial graphs, depending on room type). There are also many occurrences of 'double edges,' i.e. both the edges $A \longrightarrow B$ and $A \longleftarrow B$ exist. We break these cycles by choosing only one of the edges. If the categories of A and B are different, we orient the edge from largest-to-smallest object. Otherwise (if the categories are the same), we prefer *front* edges over *back* and *right* over *left*.

Data-driven pruning: Deleting the above edges still leaves the graph with too many relationships. To address this issue, we look to the statistics of our large scene dataset: a relationship $A \longrightarrow B$ is significant if it occurs in a significant percentage of scenes in which objects of category A and B both occur. Otherwise, we prune the edge. We use separate, increasing percentage thresholds for each distance level (3% for *adjacent*; 8% for *proximal*; 30% for *distant*), reflecting the intuition that e.g. a distant relationship must occur much more commonly than an adjacent relationship before we believe that it is intentional.

A.3 Guaranteeing Connectivity

We guarantee that extracted graphs are connected by finding all unreachable nodes and reconnecting them to the graph by searching in the original, unpruned graph for the minimum-cost path from any wall to that node. To define the cost of a path, we say that any path which induces a cycle in the graph has a higher cost than any path that does not; otherwise, the cost of a path is the sum of its edge costs. We prefer edges that are already in our pruned graph, followed by *adjacent* edges and then *proximal* edges (provided they flow from larger $\longrightarrow$ smaller objects), and finally *distant* edges (with shorter distances preferred). Due to this cycle-avoidance behavior, our final extracted graphs are acyclic (with the exception of a few kitchen scenes, which are dense with chains and adjacent edges between e.g. contiguous counter segments).

B BACKTRACKING DETAILS

This appendix provides more detail about the backtracking search procedure we use to instantiate scenes. The complete source code for this procedure is also available online at https://github.com/brownvc/planit.

Our backtracking policy is to re-sample a previously-instantiated object when our object insertion models fail to find a satisfying insertion (due to collision, constraint violation, or too much overhang for second-tier objects) for an object ≈ 10 times, decreasing as the number of sample attempts at the current object increases. We backtrack to the closest object, prior in the insertion order described in Section 6.1, that has resulted in a insertion failure. To prevent repeated rejections due to the module repeatedly resampling the same bad configuration, when an insertion point is rejected, we zero out the probability around that point in the CFCN-predicted location distribution. This starts with $0.6m \times 0.6m$, and gradually increases as number of backtracks increases.

Constraint relaxation Our scheme for constraint relaxation is as follows: we maintain a *constraint violation* cost for each object (measuring how much it violates all of its adjacent relationship edges), which is computed based on the same geometric predicates and visibility computations that we described in Section 4. It starts with 0 with no violations, and caps at 1 if either the object is completely invisible/occluded or if it is more than a certain distance away from the distance threshold. If a node has more than one constraint, a total score of 1 is distributed uniformly to each. We allow no violation for the first $1/4$ of the maximum allowed steps, and allow nodes to carry a higher cost as a quadratically increasing function of the number of backtracking steps that have been performed, capped at 1. By doing so, we will start with allowing small violations to visibility and distance, followed by allowing complete violation of one of many constraints, etc. We never allow all constraints to a node to be completely violated (i.e. a cost of 1). If that happens, the instantiation process fails.