Abstract of “Learning Autoregressive Generative Models of 3D Shapes and Scenes”
by Kai Wang, Ph.D., Brown University, May 2023.

Indoor scenes and the shapes that comprise them play a central role in our daily experiences. As virtual 3D representations of these environments become increasingly important, the value of generative models capable of producing new 3D shapes and scenes also grows. In this dissertation, we explore the possibilities of combining deep learning with large datasets of shapes and scenes to create powerful generative models of 3D shapes and scenes. We focus on autoregressive generative models, models that sequentially predict the individual components until completion. We make the following key contributions: (1) we formulate autoregressive generation of shapes, room-level scenes, and floor-level scenes: representing inputs, factorizing outputs, combining individual components and improving compatibility between these components; (2) we introduce a two-level paradigm that separate high-level semantics from low-level details, making the models more interpretable and controllable; (3) we explore strategies to encourage these models to learn more generalizable rules. Together, these contributions enable the design of generative models that generate 3D shapes and scenes with better quality, complexity, diversity, and controllability.

Learning Autoregressive Generative Models of 3D Shapes and Scenes

by

Kai Wang

B. A. Williams College, 2016

Sc. M., Brown University, 2019

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2023

This dissertation by Kai Wang is accepted in its present form
by the Department of Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____
Daniel Ritchie, Advisor

Recommended to the Graduate Council

Date _____ _____
James Tompkin, Reader

Date _____ _____
Srinath Sridhar, Reader

Date _____ _____
Manolis Savva, Reader

Date _____ _____
Angel X. Chang, Reader

Approved by the Graduate Council

Date _____ _____
Thomas A. Lewis
Dean of the Graduate School

Vita

Kai Wang was born and raised in Hangzhou, Zhejiang, China. He graduated from Hangzhou Foreign Languages School in 2011. Afterwards, he received a B.A. with honors in computer science from Williams College in Williamstown, Massachusetts, in 2016. He then pursued a Ph.D. in computer science at Brown University in Providence, Rhode Island, under the advisement of Professor Daniel Ritchie. His research revolves around 3D shapes and scenes: bridging computer graphics and machine learning, he designs data-driven algorithms that automate the process of generating 3D shapes and scenes, investigate strategies to make these algorithms more powerful and user friendly, and explore opportunities to apply these algorithms in related tasks in computer vision, robotics and beyond. During his Ph.D, he interned at Facebook AI Research and Adobe Research. He is also a recipient of the Adobe Research Fellowship in 2020.

Acknowledgements

Being able to sit here today and write the final section of my dissertation is a great privilege, and I owe my gratitude to many for their support and encouragement.

I would like to begin by thanking my mentors during my time at Brown: First and foremost, my advisor, Daniel Ritchie, for always being supportive during the highs and lows of my PhD, without which I likely would not have had the determination to finish this long journey. I am blessed that research guidance is only a small portion of what Daniel provides—assistance with and advice on paper writing and presentation, introducing me to numerous researchers, lunches at the ramen place, and countless other ways that would be too long to list here; Manolis Savva and Angel Chang, for being constants during my PhD, for six years of weekly meetings and for being invaluable friends; James Tompkin, for the consistent support ever since I arrived at Providence, across both academic and non-academic aspects of my life at Brown; Srinath Sridhar, Ellie Pavlick, Eugene Charniak, for serving on my dissertation and research comps committees, and for all the advice that helped shape this dissertation; David Laidlaw, for providing me with the opportunity for pursuing an PhD at Brown, and for all the visible and invisible support, even if I did not end up pursuing my PhD in visualization; Jeff Huang, for all the "insider" knowledge from the research method class—they really are helpful; Lauren Clarke, for all kinds of help that I cannot summarize into a sentence; Last but not least, Vova Kim, Siddhartha Chaudhuri, Minhyuk Sung, Paul Guerrero, Niloy Mitra, for mentorship and assistance with various projects.

One does not randomly decide to pursue a PhD, and I am thankful for everyone that impacted me to make this decision. I am grateful to all my teachers during my school years. Especially: Yingzi Geng, Caimei Qiu, Huiping Xie, Guming Xia, for guiding me through the tumultuous process of growing up; Jun Pan, Hailian Zou, Harvey Monaghan, for teaching me how to think mathematically and scientifically; Loke Wing Fatt, for teaching me so much about critical thinking and public speaking. Although I grew up with a strong interest in computer science, I came to college thinking about pursuing a major in philosophy and social sciences. I am

grateful towards all my undergrad professors at Williams College that convinced me to “return” to computer science: Morgan McGuire, for showing me how cool computer graphics is; Brent Heeringa, for advising my undergraduate thesis; Duane Bailey, Stephen Freund, William Lenhart, for teaching me so much about different aspects of computer science; Andrea Danyluk, for being the guiding star on my road towards PhD. Your excellent mentoring through my independent study and summer research was the deciding factor. How I wish I could share my accomplishments with you, in person, at this moment. Outside of computer science, I am also extremely grateful to all the other professors who taught me so much about other subjects, which proves to be invaluable due to the interdisciplinary nature of my PhD research, especially: Ed Gollin and Ileana Perez Velazquez, for the excellent sequence of courses on music theory; Aida Laleian and Megan Mazza, for all the memories in the photography studio; Joe Cruz and Kris Kirby, for empowering me with the lens of cognitive science, which is ever more valuable today.

I should also thank my fellow colleagues and friends: Kenny Jones, Theresa Barton, Xianghao Xu, Aditya Ganeshan, Arman Maesumi, for forming such a supportive and lively research group; Yu-An Lin, Ben Weissmann, Leon Lei, Natalie Lindsay, Ellen Jiang, Adam Wang, Leo Ko, Adrian Chang, Paul Biberstein, Ann Truong, for being an integral part for many research projects which I am or was involved in. Qian Zhang, Rao Fu, Aaron Gokaslan, for all the office chats about research and (perhaps more frequently) other things; Jingling Li, for all your support through the years, for all the gifts inside and outside Animal Crossing; Ruolan Tang, Sijia Sheng, Zihan Ling, for lunch, afternoon tea, dinner and everything else that’s not food related; Jiawei Li, Chuqing Jin, Xinyue Gu, Bingling Chen, Yuxiang Xie, Chuhang Zou, Wenjie Zhang, Jeffrey Wang, Ziqi Lu, Zhiqi Li, and all the other brilliant PhDs that I am fortunate to know for decades, it feels comforting even just knowing that someone is taking a similar path; Tianmu Lan, Yufan Fei, Baichuan Huang, Yiquan Xu, Yichen Chai, Chang Liu, Zeyuan Shang, Zhoutong Zhang, Fumeng Yang, Jing Qian, and many other wonderful people I have met during my time as a PhD student; Yuchen Huang, Ruoqi He, Ruoming Fang, Cong Chen, Lipei Li, Liqi Shu, Zihan Zhou, Xi Wang, Qingjia Jiang, Tianxia Gu, Qin Yang, Wei Sun, Ziang Zhang, Xiaoxun Jiang, Fan Zhang, Zijie Zhu, Matt Larose, Devin Gardella, Diwas Timilsina, and many more “old” friends, I wish I had more space to write more about you all.

To all the artists, musicians, game developers, chefs and athletes who make my life colorful, even during the most isolated COVID days. To all my laptops, cameras, cell phones, musical instruments, to everything that I spent my years with. To my research machine, Syenite, I wish I have utilized you for research a bit more.

To Eryi Zhou, the other me, or perhaps not after all. All my attempts in writing anything concrete about

you inevitably end up in tears, so pardon me for merely etching your existence into this dissertation. I will see you again someday, somewhere, but until then.

Finally, to all my family members, especially my parents, Yayao Wang and Junzhen Jin, for the unconditional support and love, for always standing behind me during all my major life decisions that have led me to this point. I remember your playful advice against following in your footsteps and becoming an architect. I guess, in the end, I still ended up working with indoor environments, just from a different perspective—and that doesn't feel bad at all. I hope I get a chance to return home soon and share all the stories with you in person.

Contents

List of Figures

Chapter 1

Introduction

People spend a large percentage of their lives in indoor environments. The contents of these indoor environments are naturally of great interest to us. This includes individual furniture: beds, tables, chairs, sofas, etc; rooms that contains these objects: bedrooms, living rooms, office, etc; last but not least, larger spaces containing multiple rooms: residential homes, offices, malls, etc.

As computer graphics reproduces the real world in increasing fidelity, the demand for 3D, 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 [101]. 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 [16]. Finally, machine learning researchers have begun turning to virtual environments to train data-hungry models for computer vision and robotic navigation [67, 26, 27, 43].

Given this interest in 3D virtual indoor environments, and the large amount of effort required to author them with traditional tools, a generative model of such scenes would be valuable. Many researchers have studied this problem of *generating 3D shapes and scenes* over the past decade. Early works towards this problem [4, 78] are often rule-based: they identify the design principles for 3D shapes and scenes, and encode them into cost functions which are used to optimize the individual components of shapes and scenes into desirable configurations. While effective, it requires human effort to identify the rules for specific scenarios. Such processes are time consuming, not really adaptable to other scenarios, and often fail to encode everything underlying principles that guides the generation of shapes and scenes. A natural solution to this problem is introducing machine-learning-based components that automatically extract rules from available

datasets. However, due to the limitation of dataset sizes and the capacity of the machine learning algorithms, early works either focuses on small-scale regions [34], or requires additional human inputs [39].

With the increasing availability of larger datasets of 3D shapes and scenes [114, 37, 14, 82, 12], and the emergence of deep-learning as a dominating paradigm for machine learning algorithms, new opportunities have arisen for models that can learn more rules from large amount of data. In this dissertation, we explore the possibilities for combining deep learning with large datasets of shapes and scenes, in order to create powerful models that can generate high quality 3D shapes and scenes. Specifically, we focus on a paradigm called *autoregressive generative models*, which are models that generates output sequentially by predicting and adding components one by one, conditioned on the components that has already been predicted. We explore this autoregressive generation process in a natural progression: combining (shapes) parts into shapes, shapes into room-level scenes, and finally, room-level scenes into floor-level scenes. We ask the following central question:

How to design autoregressive generative models that generate 3D shapes and scenes with better quality, complexity, diversity and controllability?

We explore this question from three main angles:

- (Angle 1) How to formulate autoregressive generation of 3D shapes and scenes?
- (Angle 2) How to make autoregressive generative models of 3D shapes and scenes more interpretable and controllable?
- (Angle 3) How to encourage the generative models to learn rules that can generalize well?

1.1 Contributions

This dissertation is based primarily on five previous conference publications [125, 100, 124, 126, 123], which introduced a series of generative models for 3D shapes and scenes. We first list these publications, in chronological order:

1. **Deep Convolutional Priors for Indoor Scene Synthesis** (Deep Synth, Figure 1.1) [125]: We present the first deep-neural-network-based method for synthesizing room-level 3D scenes. We show that deep convolutional neural networks (CNNs) can be used to effectively capture the patterns in an existing scene, without having to manually rules and relationships, as is the case for earlier works. We propose a

novel, semantically-enriched image-based scene representation based on orthographic top-down views, which CNNs can operate on. Subsequently, we learn a series of object placement priors that allows us to iteratively generate rooms from scratch, given only the room architecture as input.

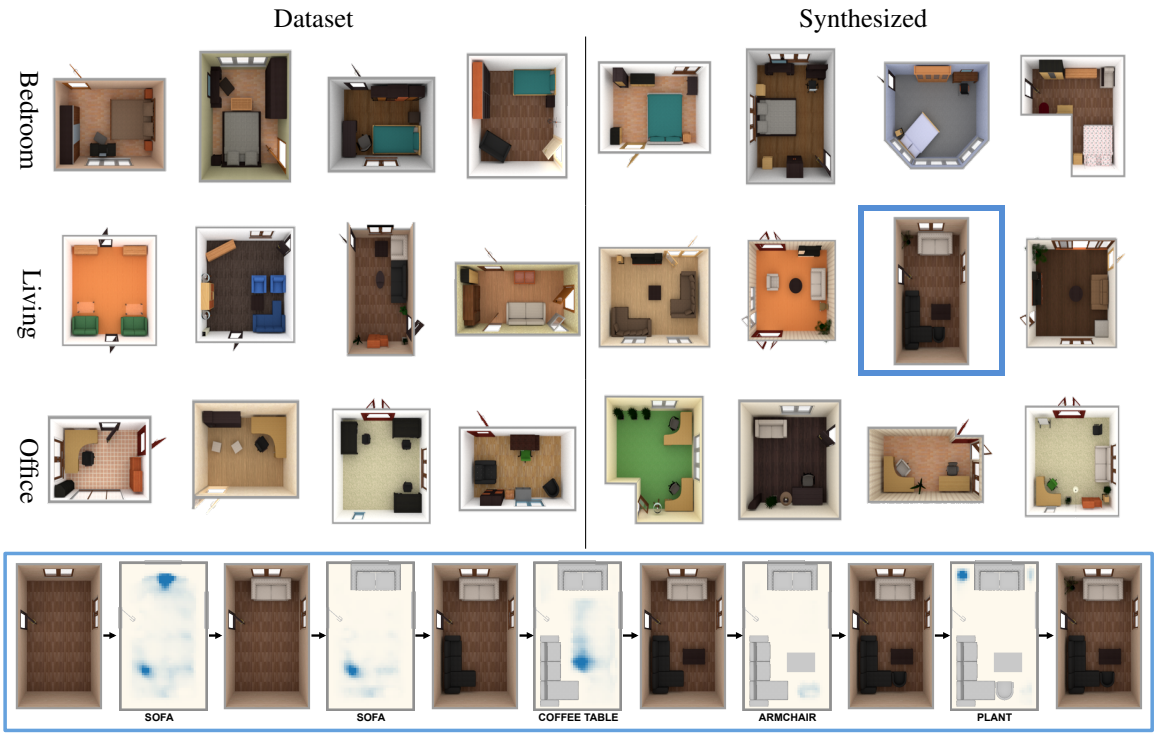


Figure 1.1: We present Deep Synth [125], a method to synthesize indoor scenes using deep convolutional network priors trained on top-down views of scenes from a scene dataset. *Top:* Bedroom, living room, and office scenes from our dataset, along with scenes synthesized from our model. *Bottom:* Our model generates scenes by iteratively inserting one object at a time. Here we show one such object insertion sequence for the living room scene outlined in blue. Renders of intermediate scenes are interleaved with a visualization of our model's predicted spatial probability distribution for the type of object about to be inserted.

2. **Fast and Flexible Indoor Scene Synthesis via Deep Convolutional Generative Models**

   (Fast Synth, Figure 1.2) [100]: We improve our previous CNN-based model by proposing a new paradigm for inserting an object into an existing scene. We use four separate neural network modules to predict the category, location, orientation and size of the object to be inserted. This new paradigm drastically reduces the amount of predictions needed for inserting a single object, making the process of generating a scene much more efficient computationally. We also show that this new paradigm allows the network to learn the distribution of existing scenes more accurately, thus improving the overall quality of generated scenes as well.

Figure 1.2: Synthetic virtual scenes generated by Fast Synth [100]. Our model can generate a large variety of such scenes, as well as complete partial scenes, in under two seconds per scene. This performance is enabled by a pipeline of multiple deep convolutional generative models which analyze a top-down representation of the scene.

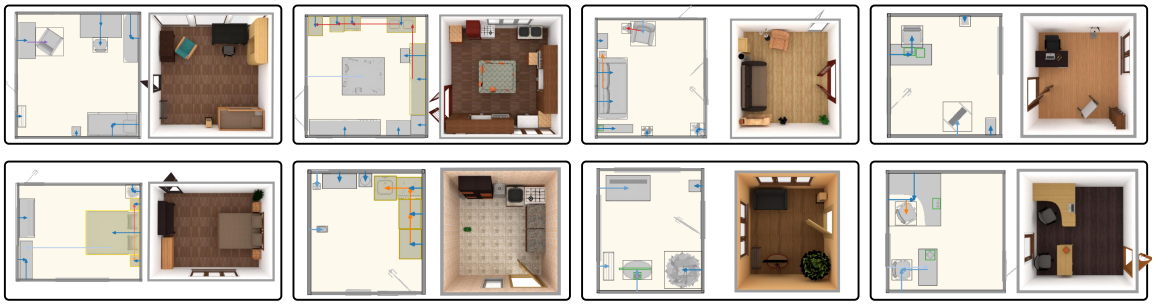


Figure 1.3: We present PlanIT [124]: a scene synthesis framework that unifies high-level *planning* of scene structure using a generative model over relation graphs with lower-level spatial *instantiation* using neural-guided search with spatial neural network modules. We first generate a relation graph 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.

3. **PlanIT: Planning and Instantiating Indoor Scenes with Relation Graph and Spatial Prior Networks** (PlanIT, Figure 1.3) [124]: We introduce a new framework that decomposes indoor scene synthesis into two phases: a high-level *planning* phase that determines of how a room should be laid out, as well as a low-level *instantiation* phase that computes the exact spatial placement of objects. We model the *planning* phase with a relations graph that defines spatial and semantic relations between objects. We propose a new graph-neural-network based generative model for such relation graphs. We modify our previous CNN-based scene generative model that predicts how an object should be placed into the scene based on the information provided with the relation graph. By decomposing the problem in this

way, we greatly improve the controllability and flexibility of indoor synthesis models, and enable a wide range of new applications.

4. **Roominoes: Generating Novel 3D Floor Plans From Existing 3D Rooms**

   (Roominoes, Figure 1.4) [126]: We propose a new task of generating novel 3D floor plans from existing 3D rooms. We identify three sub-tasks of this problem: generation of 2D layout, retrieval of compatible 3D rooms, and deformation of 3D rooms to fit the layout. We then discuss different strategies for solving the problem, and design two representative pipelines: one uses available 2D floor plans to guide selection and deformation of 3D rooms; the other learns to retrieve a set of compatible 3D rooms and combine them into novel layouts. We design a set of metrics that evaluate the generated results with respect to each of the three subtasks and show that different methods trade off performance on these subtasks. Finally, we survey downstream tasks that benefit from generated 3D scenes and discuss strategies in selecting the methods most appropriate for the demands of these tasks.

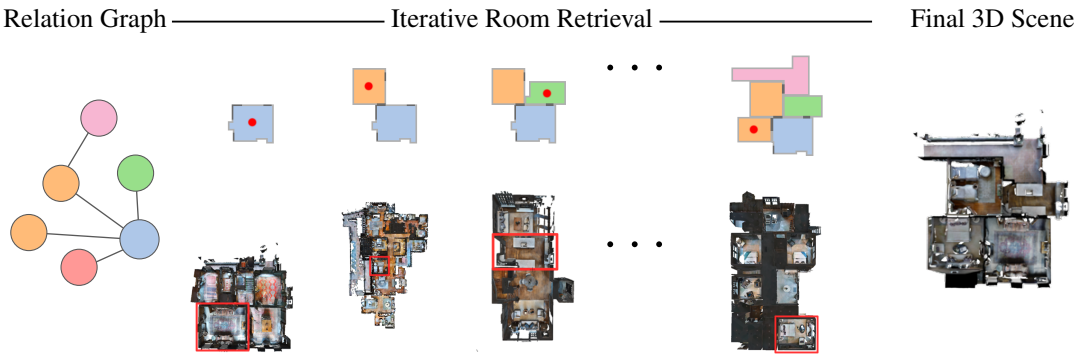


Figure 1.4: We present Roominoes [126], a new task for creating house-level 3D environments by piecing together existing 3D rooms. We devise a spectrum of potential strategies to solve this task. In an instance of this task, one of our proposed algorithms creates a combinatorially novel layout by iteratively retrieving and assembling rooms from different 3D floor plans, deforming each room as needed.

5. **The Shape Part Slot Machine: Contact-based Reasoning for Generating 3D Shapes from Parts**

   (the Shape Part Slot Machine, Figure 1.5) [123]: We present a new method for assembling novel 3D shapes from existing parts by performing contact-based reasoning. Our method represents each shape as a graph of "slots," where each slot is a region of contact between two shape parts. Based on this representation, we design a graph-neural-network-based model for generating new slot graphs and retrieving compatible parts, as well as a gradient-descent-based optimization scheme for assembling the retrieved

parts into a complete shape that respects the generated slot graph. This approach does not require any semantic part labels; interestingly, it also does not require complete part geometries—reasoning about the regions where parts connect proves sufficient to generate novel, high-quality 3D shapes. We demonstrate that our method generates shapes that outperform existing modeling-by-assembly approaches in terms of quality, diversity, and structural complexity.

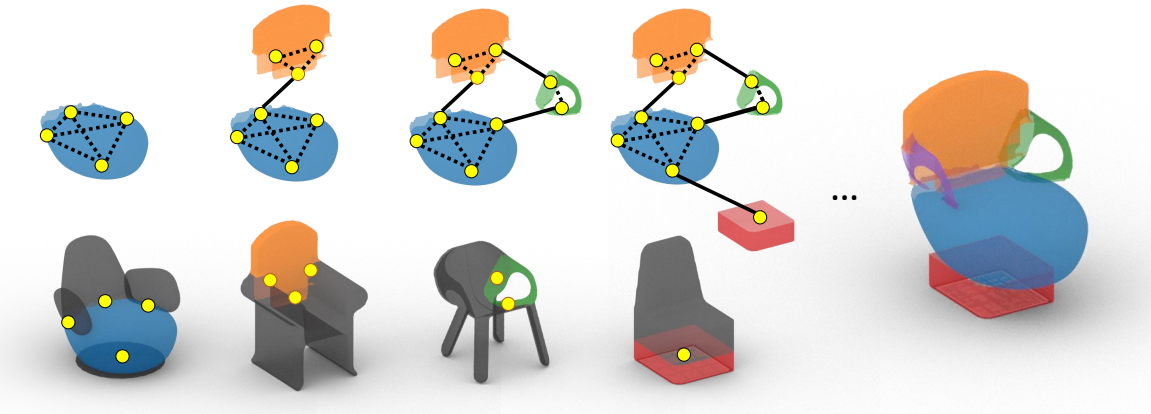


Figure 1.5: We present the Shape Part Slot Machine [123], a system that synthesizes novel 3D shapes by assembling them from parts. Internally, it represents shapes as a graph of the regions where parts connect to one another (which we call slots). It generates such a graph by retrieving part subgraphs from different shapes in a dataset. Once a full graph has been generated, the system then optimizes for affine part transformations to produce a final output shape.

Together, these five publications (for the rest of this section, numbers in parentheses refer to the corresponding publication as listed above) answer the central question of this dissertation from all three angles:

- (Angle 1) We formulated autoregressive generation of 3D shapes (5), room-level scenes (1,2,3) and floor-level scenes (4) by:
  - Proposing a set of novel **input** representations: multi-channel top-down orthographic projection of scenes (1,2,3), graphs encoding structural relationships between individual components (3,4,5). These representations allow us to leverage the power of deep neural networks and create generative models that can handle larger and more complex scenes.
  - Exploring ways to formulate the **output** of the autoregressive generative models. We show that it is possible to greatly reduce the computational load of the generative model by choosing the right type of output factorization (2), and by better utilizing the already predicted information (3).

  - Developing **optimization** strategies to combine the individual components together in plausible ways: a mixed-integer programming based procedure to deform and combine individual rooms into plausible floor plans (4), an outline-driven procedure to deform individual 3D rooms based on computed dense correspondence (4), and a gradient-descent based framework for attaching parts of a shape together given predicted part connectivity (5).
  - Improving the **compatibility** between individual components with deep metric learning, allowing the models to retrieve 3D rooms that can be stitched together into floor-plans with less amount of deformation (4), and 3D object parts that connect with each other in a more realistic and physically plausible way (5).

- (Angle 2) We make autoregressive generative models of 3D shapes and scenes more interpretable and controllable by:

  - Demonstrating that formulating generation of 3D shapes and scenes autoregressively naturally yields a set of priors for inserting a single component (1), and simplifies tasks such as completing a partial scene (2).
  - Introducing a two-level paradigm that separates high-level semantics of a scene from its low-level details (3). Such a paradigm allows users to control the output of the generative model in a more flexible way, through providing a graph, either partial or complete, that describes the type of desired high-level constraints.
  - Extending this idea to 3D shapes (5) and floor plans (4), adapting similar strategies to control the output with a graph specification.

- (Angle 3) We encourage our generative models to learn more generalizable rules by:

  - Ensuring that the models are not overfitting to single training examples and encouraging them to create continuous output distributions (1,2).
  - Using input representations that highlights the most important scene attributes (1,2), and isolates the high-level semantics of a scene (3).
  - Proposing a novel “slot”-based representation of 3D shapes (5) that facilitates the model to perform contact-based reasoning and combining shape components in new ways.

By exploring these angles, we make important steps towards achieving the goals listed in the central question of this dissertation. We show that the generative models proposed by this dissertation indeed generate outputs with better:

- **Quality**: We provide qualitative examples for shapes and scenes generated by our methods, and analyze why they compare favorably than those generated by prior works and baselines (1-5). We support our analysis with a variety of quantitative metrics: human perceptual studies (1-3), distributional similarity (1-5), geometry quality (4), physical plausibility (5) and performance on downstream tasks (4).
- **Complexity**: We show qualitatively that our methods can handle more complex shapes and scenes than prior works (1,5), and that they use more components on average (5). We also show that our methods can work in a wider range of scenarios, extending support to irregular room geometries (1-3) and second-tier objects (2,3).
- **Diversity**: We demonstrate that our methods can generate multiple possible outputs given the same (partial) input (1-3,5), can combine components from multiple input examples into novel outputs (1-5) that are noticeably different then the nearest neighbors in the training set (1,2,5).
- **Controllability**: We show that modules used by our methods generate plausible output distributions useful for interactive tasks e.g. predicting where to place an component (1-3), finding most compatible components (4,5). We demonstrate that our methods can support a wide range of editing tasks e.g. completing partial 3D scenes (2), generating 3D scenes from high level descriptions (3).

1.2 Dissertation Overview

The rest of the dissertation is organized as follows:

In Chapter 2, we formally define autoregressive generative models of 3D shapes and scenes and discuss related works that concern generative models of shapes and scenes.

In the next five chapters, we present our contributions in more detail, with each chapter corresponding to one of the publications that form this dissertation. Chapter 3 describes Deep Synth, our deep CNN based method for indoor scene synthesis. Chapter 4 describes Fast Synth, our improve to Deep Synth based on a new paradigm for predicting the attributes of individual objects. Chapter 5 introduces PlanIT, a paradigm for breaking down indoor scene synthesis into high level planning and low level instantiation. Chapter 6 details Roominoes, a new task for generating novel 3D floor plans, as well as a survey of strategies for solving this

problem. Chapter 7 introduces the Shape Part Slot Machine, a new approach for assembling novel 3D shapes from existing parts.

Finally, in Chapter 8, we conclude this dissertation by summarizing our main contributions and discuss possible future directions for designing and applying autoregressive generative models of 3D shapes and scenes.

Chapter 2

Background

In this chapter, we provide an overview of the most relevant background material to our technical contributions. We begin by formally defining the problem we study in this dissertation, namely, learning autoregressive generative models of 3D shapes and scenes. We then survey the approaches for generation of shapes and scenes, both in 2D and 3D.

2.1 Definition

In this dissertation, we focus on learning autoregressive generative models for 3D shapes and scenes. In this section, we clarify the scope of the problem by specifying the types of 3D shapes and scenes we consider and providing a precise definition of autoregressive generative models in this context.

2.1.1 3D Shapes and Scenes

The world consists of a diverse array of 3D shapes and scenes, ranging from from natural landscapes to urban structures, and from biological organisms to man-made objects. In dissertation, we focuses on 3D shapes and scenes that encompass indoor environments. Specifically, we examine them on three levels: 1. indiviudal 3D objects or shapes, such as beds, tables and sofas; 2. room-level 3D scenes that contains these objects, such as bedrooms, living rooms and offices; and 3. floor-level 3D scenes that comprise multiple 3D rooms.

To learn generative models off 3D shapes and scenes, one requires a collection of 3D shapes and scenes as training data. The following datasets are used in this dissertation: 1. for 3D objects, we use the PartNet [82] dataset, which provides part-level annotation for a large collection of 3D shapes from the ShapeNet [14]

dataset; 2. for room-level 3D scenes, we use the SUNCG [114] dataset, a large database of virtual 3D rooms created by users of the online Planner5d design tool [90][1]; 3. for floor-level 3D scenes, we use the RPLAN [133] dataset, a collection of floor plan images annotated at pixel level, as well as the Matterport3D [12] dataset, a collection of scanned 3D floors.

From these datasets, we extract a *part-based* representation of 3D shapes and scenes: 1. 3D objects are represented as individual components e.g. a chair as seat, back and legs; room-level 3D scenes are represented as individual architectural elements (e.g. wall, door, window, floor) and furniture; floor-level 3D scenes are represented as a collection of individual 3D rooms.

Formally, we work with datasets $D = \{(S_1, \ldots, S_N), (P_1, \ldots, P_M)\}$ that contains a collection of shapes or scenes S and a collection of individual components P. Each shape or scene in the dataset $S_i = \{(P_1, T_1), \ldots (P_O, T_O)\}$ is composed of a collection of components P, along with transforms T that transform these components so they fit into the shape or scene. In the datasets used by this dissertation, each transform T is limited to rigid transformation and scaling, either isotropic or anisotropic.

2.1.2 Learning Autoregressive Generative Models

Given a dataset D, our goal is to learn a generative model that maximizes the likelihood of the training samples, and thus being able to generate novel 3D shapes and scenes. Formally, we aim to find a parameterized model $p_\theta(S)$ that maximizes the probability of the training samples D:

$$\theta^* = \arg\max_\theta \sum_{i=1}^{N} \log p_\theta(S_i), \tag{2.1}$$

where θ denotes the model parameters, and S_i represents a shape or scene instance from the training set.

There are many ways to design such generative models. In this dissertation, we focus on a specific class of generative models: autoregressive generative models, which have been successfully applied to various domains such as 2D image generation [31, 44, 46, 120, 96], language models [11, 145], audio synthesis [119], procedural modeling [99], 3D shape generation [155, 111, 117], etc. Instead of estimating the probability of a shape or scene S directly, autoregressive models decompose it into a chain of conditional probabilities:

$$p_\theta(S) = \prod_{i=1}^{N} p_\theta(P_i, T_i \mid S_{<i}), \tag{2.2}$$

[1]This dataset is no longer available due to an ongoing legal dispute. The part of the dissertation that utilizes this dataset was finished and published before this incident. No new results on this dataset has been produced after that.

where $S_{<i} = (P_1, T_1 \dots P_{i-1}, T_{i-1})$ is the partial shape or scene at the current step. That is, the model estimates the probability of the i-th component and its transform by conditioning on all the previous components. Note that the ordering of the components can be arbitrary, which makes the model permutation invariant. In other words, an autoregressive generative model generates a new shape or a scene by iteratively selecting and adding components, conditioning each new component based on the already generated *partial* shape or scene.

To add a new component (P_i, T_i), we further decompose it into a set of sub-attributes and estimate these attributes autoregressively:

$$p_\theta(P_i, T_i \mid S_{<i}) = \prod_{j=1}^{M} p_\theta(A_{ij} \mid A_{i1}, \dots, A_{i(j-1)}, S_{<i}), \tag{2.3}$$

where A_{ij} represents the j-th sub-attribute of (P_i, T_i). There are many strategies for decomposing the components into sets of sub-attributes. In this dissertation, especially in Chapter 3 and 4, we propose various decomposition strategies and also identify a suitable ordering for estimating the decomposed sub-attributes.

Starting from Section 5, we also explore two-pass strategies where we first autoregressively generate an intermediate representation of the shape or scene. In our approaches, this intermediate representation takes the form of a relation graph $G = (V, E)$, where $v \in V$ corresponds to the components and the edges $e \in E$ capture the relations between components. We either use ground truth relation graphs (Section 6) or generate them autoregressively (Section 5, 7), inspired by recent developments in graph generative models [63, 70, 148]. We discuss the detailed formulations in the respective sections. We then condition on this intermediate relation graph to generate the full shape or scene in the same autoregressive fashion.

2.2 Related Generative Models of Shapes and Scenes

Generation of shapes and scenes is a problem that has long been a problem of interest to many fields. While the interest for the problem has traditionally been in computer graphics and architecture, in recent years, the problem has also drawn the interests of researchers in computer vision and robotics, who has a strong demand for such shapes and scenes. Researchers in machine learning have also been paying increasingly more attention to this problem, as the combinatorial complexity of shapes and scenes make them an interest class of data for learning. In this section, we briefly survey works that generate shapes and scenes at various scales, employing a range of representations in both 2D and 3D. For a more detailed survey of such methods,

readers are encouraged to refer to the survey by Chaudhuri et al. [19].

2.2.1 Non-part-based Generation of Shapes and Scenes

As mentioned in the previous section, we focus on *part-based* generation of 3D shapes and scenes. There also exist work to do not perform such part-based modeling. One popular approach is directly generating an 2D, image-based representation of a scene, using one of the popular image generative models such as Generative Adversarial Network [42] or Variational Autoencoders [62]. In addition to direct generation, works have also looked into manipulating the generated images based on various types of constraints [142, 75]. It is also possible to generate 3D shapes and scenes with non-part-based 3D representations, such as voxels [134, 130], point clouds [143, 116], meshes [83], surfaces [45], and implicit functions [87, 23, 79]. These works, however, are usually not aware of the structural information of 3D shapes and scenes present in a more part-based representation. For the rest of this section, we focus on methods that perform part-based generation of shapes and scenes.

2.2.2 Generative Models of 3D Shapes

Many works have adopted a part-based strategy for generating 3D shapes. One option is to make voxel-grid generative models part-aware [122, 135]. Many models have been proposed which generate sets of cuboids representing shape parts [155]; some fill the cuboids with generated geometry in the form of voxel grids [68] or point clouds [81, 56, 57]. Other part-based generative models skip cuboid proxies and generate part geometries directly, as point clouds [107], implicit fields [131], or deformed genus zero meshes [40, 144]. All of these models synthesize part geometry, but such synthesized geometries often contain artifacts, and do not match real world, available parts. Instead of generating geometry, one can also assemble shapes from existing high-quality part meshes. The Modeling By Example system pioneered the paradigm of modeling-by-assembly with interactive system for replacing parts of an object by searching in database [39]. The Shuffler system added semantic part labels, enabling automatic 'shuffling' of corresponding parts [66]. Later work handled more complex shapes by taking symmetry and hierarchy into account [54]. Other modes of user interaction include guiding exploration via sketches [137] or abstract shape templates [5], searching for parts by semantic attributes [17], or having the user play the role of fitness function in an evolutionary algorithm [140]. Probabilistic graphical models have been effective for suggesting parts [18] or synthesizing entire shapes automatically [59]. Part-based assembly has also been used for reconstructing shapes from

images [112]. The method proposed in this dissertation [123] is most closely related to ComplementMe, which trains deep networks to suggest and place unlabeled parts to extend a partial shape [117]. Our model is different in that we use a novel, part-contacts-only representation of shapes, which we show enables handling of more structurally complex shapes.

2.2.3 Generative Models of Room Level Scenes

There is a large body of existing work on synthesizing room level interior scenes. Some of the early work in this space is concerned with creating plausible arrangements of a pre-specified set of objects according to interior design principles [78] and simple statistical relationships between objects learned from examples [149]. Later work showed how to arrange objects in scenes where the set of objects to be used can change, using a hand-written program to specify priors over which objects may occur and their spatial relationships [146]. Follow-on work from this has focused on *data-driven scene synthesis*: rather than using a hand-written program, learning priors over object occurrence and arrangement from examples. The first such method learned separate priors for occurrence and arrangement, using a directed graphical model of object co-occurrence relationships and a Gaussian mixture model of pairwise spatial relationships between objects [34]. Various related methods have been proposed, modeling object occurrence and/or arrangement with undirected factor graphs [60], topic models [71], directed graphical models combined with Gaussian mixture arrangement patterns [48], human-centric stochastic grammars [92], and activity-based object relation graphs [38]. Other related work has focused on generating virtual indoor scenes given some sparse or low-fidelity input representation. Methods have been proposed for generating scenes from sketches [139], from noisy 3D scans [35], or given text descriptions [15], all of which extend scene synthesis methods from the list above [34].

In contrast, this dissertation uses deep convolutional networks to learn priors over which objects should be in a scene and how they should be arranged. Through three previous publications [125, 100, 124], we explore different ways to formulate autoregressive generation of room level scenes, by proposing different ways to predict the attributes needed to insert an individual object [125, 100], as well as introducing a two-level pipeline that separates high-level properties of a room from low level details [124].

2.2.4 Generative Models of Floor Level Scenes

There is limited work that attempts to generate 3D environments with multiple rooms. Procedural content generation for studying RL is starting to leverage generated 3D environments. However, these are mostly

restricted to 3D mazes [7] or towers with rooms [58], which do not reflect the complexity of real-world furnished houses. A related line of work addresses procedural generation of floor plan layouts for residences and other buildings [77, 6, 73]. These works, however, focus on generating the 3D architectural structure and do not typically produce realistically furnished 3D interiors. In contrast, there is a large body of existing work on generating 2D floor plans. As part of a larger system for generating 3D residential house models, Merrell et al. introduced a Bayesian network for modeling the room relationship graph along with an optimization-based approach for realizing this graph through precise wall geometry [77]. Liu et al. also use an optimization-based approach as part of a machine-assisted interactive system for designing precast concrete buildings [73]. More recently, Wu et al. developed a mixed-integer quadratic programming (MIQP) formulation for optimizing building interior layouts [132]. Finally, the past year has seen the introduction of deep learning methods into this problem space, resulting in learning-based methods for generating floor plans given building outlines [133], room relationship graphs [84, 85], or both [50]. These methods all address generation of 2D layouts with unconstrained room shapes. Very recent work takes a different approach of learning to generate a constraint graph and then solving the optimization problem posed by this graph to produce a final layout [86]. In contrast to all these approaches, this dissertation seeks solve the problem of generating new *3D* layouts through combining existing 3D rooms, where the room shapes are constrained by an available set of existing 3D rooms [126].

Chapter 3

Deep Convolutional Priors for Indoor Scene Synthesis

One of the key challenges in learning generative models of 3D scenes is extracting and modeling the various types of relationships between individual objects in a scene.

Prior work has either focused on modeling smaller, functional regions within a larger room (i.e. a working desk) [34], or introduced additional inputs to constrain the problem: a fixed set of objects and manually-annotated important relationships [149], a sketch [139], a natural language description [15], or a 3D scan of a room [21, 35].

The key insight of this chapter is that convolutional neural networks (CNNs) can serve as a powerful tool for modeling these relationships, without having to resort to manual annotation and handcrafted rules. CNNs have been demonstrated to reliably learn to recognize and generate visual patterns and relationships in other graphics domains. Using large-scale datasets of 3D scenes datasets [114], we train train a CNN-based model to select and place objects in a room given only the type of room desired and its wall structure as input.

To apply convolutional networks to the scene synthesis problem, we leverage the insight that while an indoor room is a 3D space, most objects that characterize a room are arranged on its 2D ground plane. As such, we represent a room via an orthographic top-down view, a representation on which a convolutional network can operate—just as architects are trained to think of rooms as patterns of objects on a floor plan, so too do we train our neural networks. Our top-down view representation is a semantically-enriched, multi-channel

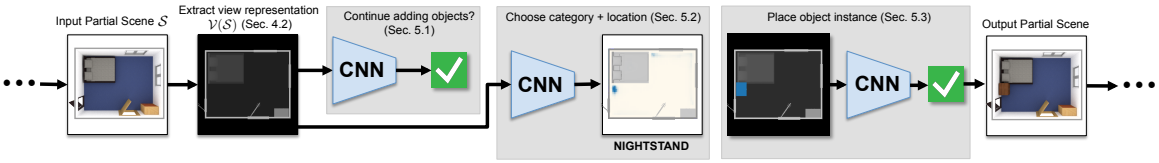


Figure 3.1: Overview of our scene synthesis pipeline. Our generative model synthesizes scenes one object at a time. Given an input scene $\mathcal{S}$, it first computes a top-down view of the scene $\mathcal{V}(\mathcal{S})$ with multiple features per pixel (Section 3.2.1). Next, it analyzes this image to determine whether to add another object (Section 3.3.1). Then, it chooses a location at which to add the next object, and what category of object to add (Section 3.3.2). Given a location and category, it selects a instance of that category from a model database and adds it to the scene at an appropriate orientation (Section 3.3.3). These model components are implemented using deep convolutional networks.

image capturing multiple scene features per pixel, such as the type of object present and its orientation. Applying layers of learnable convolutions to this representation allows our model to process the entire state of a room as context for its sampling decisions, capturing patterns and relationships between objects . Our proposed model generates a room by iteratively adding one object at a time, first deciding whether to add another object before deciding which type of object to place and where. Each of these decisions is governed by a convolutional neural network.

In this chapter, we make the following contributions:

1. We present the first convolutional network-based system for synthesizing 3D indoor scenes from scratch: our system takes as input only the geometry of the room in which it should synthesize a scene.
2. We introduce a new semantically-enriched, image-based representation of scenes based on orthographic top-down views.
3. Through a series of ablations, we analyze our learned convolutional priors and show they capture common-sense patterns of object arrangement given the state of a room.

We first give a high-level overview of our approach in Section 3.1. We then introduce our dataset and our image-based scene representation in Section 3.2 before describing the details of our generative model in Section 3.3. Finally, Section 3.4 presents the results of our perceptual studies. Our code and data are available at `https://github.com/brownvc/deep-synth`

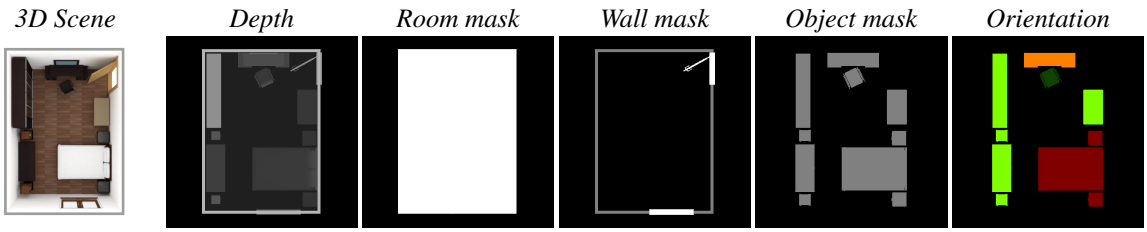


Figure 3.2: Image channels included in our top-down view representation $\mathcal{V}(\mathcal{S})$ of a scene $\mathcal{S}$. Object category channels not shown, for brevity. Orientation is visualized by normalizing $\cos\theta$ and $\sin\theta$ to $[0, 1]$ and mapping them to the red and green channels of an RGB image.

3.1 Approach

This chapter focuses on learning generative models of indoor scenes, specifically, rooms that contain one or more objects. The input to our method is a room, i.e. geometry describing floor, walls, and ceiling. In this chapter, we focus on generating and arranging major functional objects such as furniture which are placed on the floor. We do not address wall-mounted or second-tier objects (i.e. objects placed on top of other objects), though we discuss straightforward extensions to support these features in Section 3.5, and incorporate second-tier objects in the next two chapters.

Our goal is to build a generative model of scenes that can encode complex dependencies and relationships involving objects and the room geometry. We use deep networks as the major building block of our model, as they have been shown to reliably recognize and generate complex visual patterns in other visual domains. To do this, we exploit the fact that while indoor scenes exist in 3D space, gravity dictates that most objects are arranged on the 2D floor plane. Thus, we can apply 2D convolutional networks to a top-down view representation $\mathcal{V}(\mathcal{S})$ of a scene $\mathcal{S}$, where the networks learn to recognize the presence of and relationships between objects.

Since scenes are composed of a set of discrete objects, we use a model architecture that generates a scene iteratively, adding objects one-by-one. This sequential paradigm has been frequently used to model other visual content that decomposes into discrete components [46, 155, 111, 99, 29, 117]. Figure 3.1 shows a schematic overview of one iteration our method, somewhere in the middle of a synthesis sequence. Our model first takes the scene constructed thus far and renders a top-down view of it. This top-down view is augmented with multiple semantic features, such as the category of the object at each pixel (Section 3.2.1). This view $\mathcal{V}(\mathcal{S})$ of a scene $\mathcal{S}$ is an image-based representation of the scene that can be analyzed by convolutional networks (CNNs); it is used as input by several CNN-based components of our model.

The first of these components predicts whether the model should keep adding objects to the scene (Section 3.3.1). If this component returns true, the next model component chooses a category of object to add and a location at which to add it. This component works by building a probability distribution of possible object categories across many possible scene locations, then sampling from that distribution (Section 3.3.2). Given an object category and location, the model must then instantiate a particular object that belongs to that category at that location, orienting the object appropriately in the process (Section 3.3.3). These three model components all use deep convolutional networks which take the scene view as input. These networks are trained on scenes sampled from a large database of 3D scenes; Section 3.2 describes our method of constructing training data in detail. Figure 1.1 Bottom shows a scene synthesized by executing multiple iterations of this process.

3.2 Dataset

Learning a deep network-based model requires a large amount of training data. To train our model, we use the SUNCG dataset [114], a large database of virtual 3D scenes created by users of the online Planner5d interior design tool [90]. The dataset contains over 45,000 3D scenes, with each scene segmented into individual rooms labeled with a room type (e.g. bedroom, living room) or multiple room types. In this work, we consider three different types of rooms that occur frequently both in residential interior environments and in the SUNCG dataset: bedrooms, living rooms, and offices. As the SUNCG dataset is large and noisy, we first do some preprocessing by filtering out certain rooms and certain types of objects from rooms. Appendix A.1 provides more details on these filters. After filtering, we are left with $8,398$ bedrooms, $1,238$ offices, and $1,452$ living rooms. For each room type, we hold out 250 rooms for validation and testing and use the rest for training.

3.2.1 Top-down View Representation

As mentioned previously, indoor rooms are largely characterized by the 2D layout of objects on the floor; much prior work on scene synthesis phrases the problem in terms of determining 2D positions and 1D orientations of objects on a supporting surface [149, 78, 34]. Likewise, we convert 3D rooms into a 2D layout representation that our model operates on using deep convolutional networks. Specifically, we represent the scene $\mathcal{S}$ as a multi-channel top-down view $\mathcal{V}(\mathcal{S})$. The base of this representation is an orthographic top-down depth render of the room, i.e. a heightmap. This render maps a $6\,\mathrm{m} \times 6\,\mathrm{m}$ region into a 512×512 image.

Discrete-sampled heightmaps have been used for scene synthesis before, as a coarse representation of the geometry of a scanned 3D scene that synthesis should match [35].

With this height map alone, the deep networks in our model would be forced to use some of their representational capacity to learn how to detect semantic features in the height map, such as the boundaries, orientations and categories of objects (and this may or may not succeed). Instead, we encode such semantic features as additional channels in the top-down view image. The final image has the following information at each pixel, which defaults to 0:

Depth: Depth from the camera.
Room mask: Takes a value of 1 when inside the room.
Wall mask: Takes a value of 0.5 for walls and 1 for doors/windows.
Object mask: Indicates the number of objects present. Each object adds 0.5 to the pixel value.
Orientation: The orientation of the object contained by the pixel, represented by an angle θ about the world-up vector. θ is expressed in a local coordinate frame that is consistent across all objects. We encode this information into the image using two channels, one for $\sin\theta$ and one for $\cos\theta$ (i.e. a polar-to-rectangular transform).
Category: The category of the object(s) contained by the pixel. For each category, including doors and windows, we add a channel that stores the number of objects of that category.

Figure 3.2 shows what some of these channels look like for a typical room. This representation has C + 6 channels total, where C is the number of possible categories for the type of room. In our dataset, C ranges from 21 (living room, office) to 31 (bedroom). Though we do not experience hardware or training problems with this representation, for significantly larger C, it may become necessary to instead learn a fixed-dimensional representation for object categories, serving a similar function as word embeddings in natural language data [80]. We leave such an effort for future work.

3.3 Generative Model

Our model generates a scene $\mathcal{S}$ by iteratively placing one object at a time, where the existence and configuration of the object being placed is conditioned on the state of the scene thus far. Each iteration of our model decomposes into three steps: deciding whether to add another object (`Continue?`), deciding what category

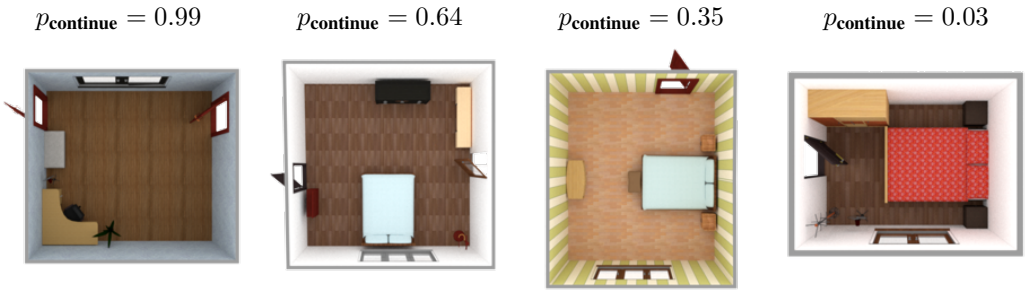


Figure 3.3: Predicted $p_{\textbf{continue}}$ values for partial bedroom scenes from our dataset. From left to right: the absence of a bed leads to continue probability near 1.0, a scene with a bed has lower continue probability but still above 0.5, a scene that is complete enough to terminate but could still have more objects added, a scene that is completely full and thus has nearly 0 continue probability.

of object to add and where (`CategoryLocation`), and inserting an instance of that object category into the scene (`InstanceOrientation`).

3.3.1 When to stop adding objects

The first component of our model, `Continue?`, decides whether the model should add another object to $\mathcal{S}$ or should terminate. It is a function which takes as input the current scene $\mathcal{S}$ and outputs a Bernoulli distribution $p_{\textbf{continue}}(\top|\mathcal{S})$, i.e. the probability that 'continue adding objects' is true (i.e. logical $\top$) given the current scene $\mathcal{S}$. We encapsulate the current state of the scene by two sets of features. First, we use a vector of counts of each category of object already present in the scene, counts($\mathcal{S}$). This provides *global* information about the state of the scene that may be important for making continue/terminate decisions, e.g. a bedroom with zero beds must continue adding objects. Second, we extract high-level features from the top-down view representation of the scene $\mathcal{V}(\mathcal{S})$ using a deep convolutional network. These features provide information about whether the current set of scene objects represents a complete scene *for the given room*, e.g. while a single bed and nightstand may be sufficient for a tiny, narrow room, a large bedroom requires more objects. We then apply a multilayer feed-forward neural network (MLP) to compute the probability of continuing:

$$p_{\textbf{continue}}(\top|\mathcal{S}) = \sigma(\text{MLP}([\text{counts}(\mathcal{S}), \text{CNN}(\mathcal{V}(\mathcal{S}))]))$$

where $[\cdot]$ is vector concatenation and σ is the logistic sigmoid.

We build a training set for `Continue?` with 50% negative examples (complete rooms from our dataset) and 50% positive examples (rooms with a random number of random objects removed). Figure 3.3 shows

some partial bedroom scenes from our dataset with their predicted $p_{\textbf{continue}}$ values. In the upper left scene, the room has no bed and thus $p_{\textbf{continue}}$ is near 1. The upper right scene has a bed but is otherwise somewhat sparse; $p_{\textbf{continue}}$ is lower but still above 0.5. The lower left scene has enough appropriate objects to be reasonably considered a 'complete' bedroom, and thus has $p_{\textbf{continue}} < 0.5$, but it also has space for more objects. Finally, the lower right scene is completely packed with objects, leading to a $p_{\textbf{continue}}$ near 0.

We train `Continue?` using the standard binary cross entropy loss. At synthesis time, we terminate synthesis when $p_{\textbf{continue}}$ falls below 0.5 (i.e. the classification decision boundary).

If `Continue?` decides to add another object, our model must then decide both which category of object to add and also where in the room to add it. These two decisions are strongly correlated: some locations only make sense for certain types of objects, and vice versa. Ideally, we would like our model to learn the joint distribution $p(c, x, y|\mathcal{S})$ over all categories and all possible locations in the scene. To make this problem amenable to the use of convolutional networks, we instead learn a conditional distribution. Specifically, our next model component `CategoryLocation` computes $p_{\textbf{cat}}(c|\mathcal{S}, x, y)$, the probability that c is the category which should be added to scene $\mathcal{S}$ at location (x, y). This structure is similar to the 'action map' representation used in prior work to encode probabilities of different human activities taking place across different parts of a scene [105]. Later, we describe how we construct the joint distribution from this conditional and then sample from it.

3.3.2 What category of object to add next (and where)

To compute these probabilities, `CategoryLocation` uses a similar structure as `Continue?`: a convolutional network to extract spatial features from the current state of the room, along with the room's current category counts. Since `CategoryLocation` computes probabilities for a particular location (x, y), it must incorporate additional information to instruct the CNN to attend to that location. For this purpose, we add an additional *attention mask* channel to the top-down view image $\mathcal{V}(\mathcal{S})$. This is an image $\mathcal{M}_{\textbf{attn}}(x, y)$ containing a small (9×9) mask centered about (x, y); Figure 3.4 shows some examples. Category probabilities are then computed as

$$f_{\textbf{cat}}(c|\mathcal{S}, x, y) = \text{MLP}([\text{counts}(\mathcal{S}), \text{CNN}([\mathcal{V}(\mathcal{S}), \mathcal{M}_{\textbf{attn}}(x, y)])])$$
$$p_{\textbf{cat}}(c|\mathcal{S}, x, y) = \frac{\exp(f_{\textbf{cat}}(c|\mathcal{S}, x, y))}{\sum_{c'} \exp(f_{\textbf{cat}}(c'|\mathcal{S}, x, y))}$$

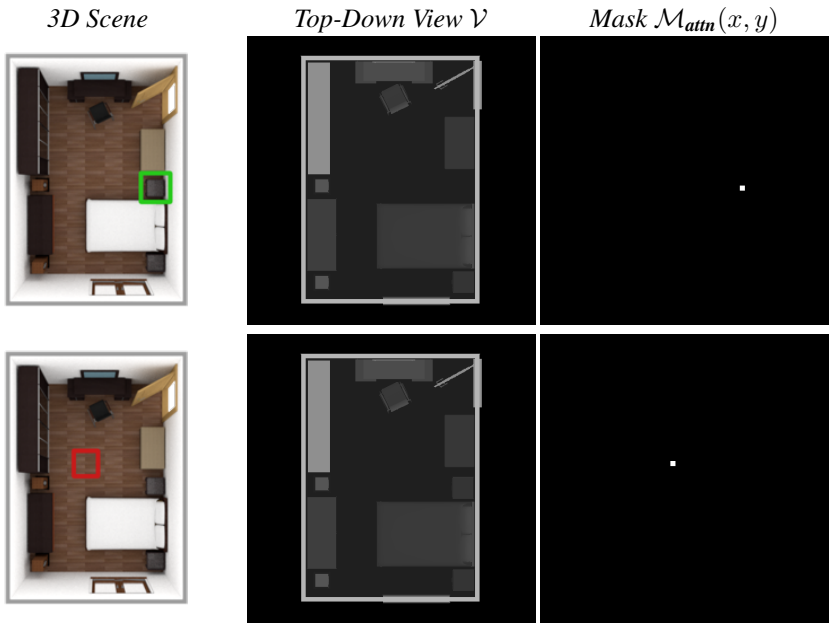


Figure 3.4: Illustrating the training examples (x, y, c) used to train `CategoryLocation`. The colored boxes are centered around the location (x, y). *Top:* an example where $c =$ 'nightstand'. The third column shows the attention mask $\mathcal{M}_{\textbf{attn}}(x, y)$. *Bottom:* an example where $c =$ 'no object'.

i.e. we compute $f_{\textbf{cat}}(c|\mathcal{S}, x, y)$ and then normalize it using a softmax activation. Figure 3.5 Top visualizes a few representative examples of these probabilities evaluated across all locations in a scene.

$\mathbf{p}_{\textbf{cat}}(c|\mathcal{S}, x, y)$ gives a probability distribution over object categories with centroid at location (x, y). As it is a distribution, it sums to one, which would imply that there is *some* object centroid at every location. This is obviously implausible: a large fraction of locations in a finished room remain unoccupied by objects, to permit human movement. Thus, we augment the set of object categories with an *auxiliary category* for 'no object centroid.' We also add auxiliary categories for 'existing object centroid' (i.e. a location in the scene already occupied by an object centroid) and for 'outside room,' to allow `CategoryLocation` to learn how such locations differ from those in which object centroids may be placed. Figure 3.5 Bottom shows probabilities for these categories across an example scene.

Training

To generate training examples for `CategoryLocation`, we take rooms from our dataset and randomly remove objects from them. To generate examples for object categories, we choose a random removed object and generate an attention mask $\mathcal{M}_{\textbf{attn}}(x, y)$ about that object's centroid (x, y). To generate examples for auxiliary categories, we choose a random point that is outside the room, in empty space within the room,

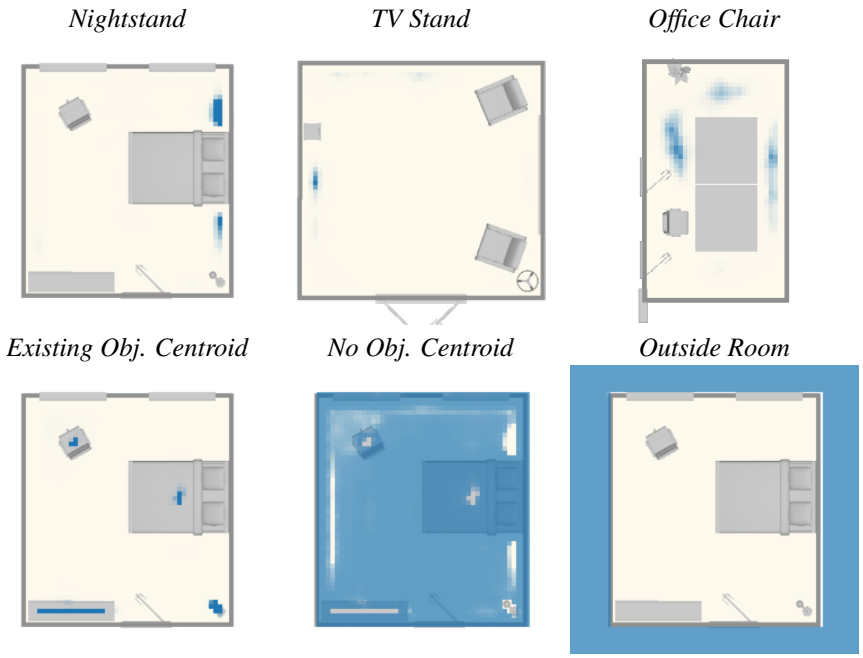


Figure 3.5: Examples of predicted category probabilities for all locations in a scene. *Top:* predicted probabilities for several object categories in different scenes. *Bottom:* predicted probabilities for auxiliary categories for the same bedroom scene.

or within an existing object, and we generate an attention mask about that point. Figure 3.4 illustrates two examples of this process. We generate training sets with 95% auxiliary category examples and 5% object category examples. This balance makes intuitive sense, given the sparsity of object centroid locations in a scene. The low proportion of object category examples can cause the network to train slowly; to speed this up, we start training with only object category examples and gradually increase the percentage of auxiliary category examples to 95%.

Given the large number of possible conditioning inputs $(\mathcal{S}, x, y)$, the network may not see enough cases to learn rarer rules, e.g. not to place a third bed in a bedroom that already has two. To help with this situation, we train `CategoryLocation` with the standard categorical cross entropy loss along with an additional loss $\mathcal{L}_{\textbf{global}}$. This loss provides additional global context by penalizing the model for assigning probability to categories not in the set $\mathcal{C}_{\textbf{removed}}$ of object categories removed from the current partial training scene. We enforce this loss more strictly for nearly-complete partial training scenes, linearly scaling it based on the ratio of the number of objects in the partial scene to that of the complete scene:

$$\mathcal{L}_{\textbf{global}} = \frac{N_{\textbf{partial}}}{N_{\textbf{complete}}} \sum p(c) \; \forall c \notin \mathcal{C}_{\textbf{removed}}$$

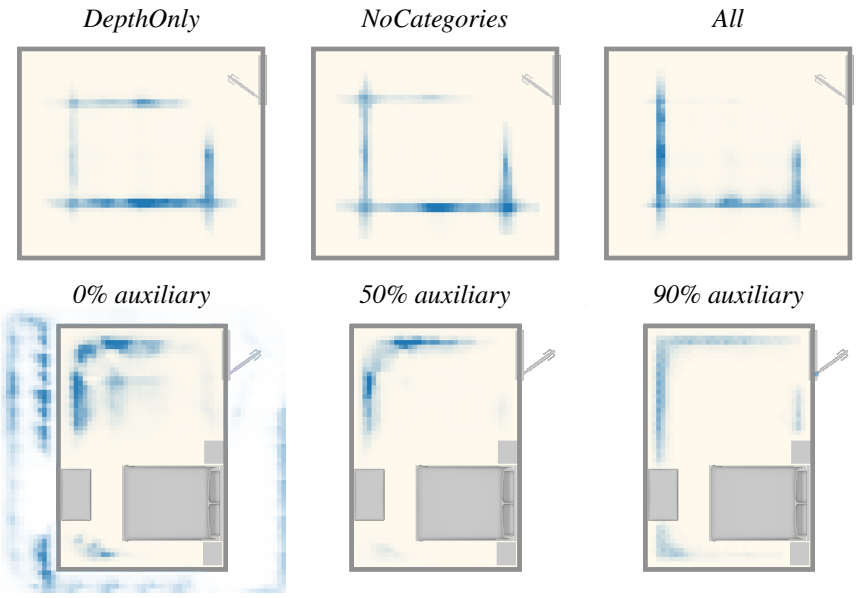


Figure 3.6: Example object category location distributions predicted by the `CategoryLocation` network with certain features omitted. *Top:* Predicted distributions for the centroid of a bed with scene view features excluded. Only when all features are present does the distribution avoid parts of the scene that would block the door in the upper-right. *Bottom:* Predicted distributions for a wardrobe, varying the percentage of auxiliary category examples in the training data. A high percentage is needed to learn a distribution that is flush against the walls.

Ablations

We also trained versions of `CategoryLocation` with features omitted from the scene view images $\mathcal{V}(\mathcal{S})$. In addition to the full set of image features (*All*), we considered scenarios with the category mask channels removed (*NoCategories*) and with all channels except depth removed (*DepthOnly*). We also experimented with the percentage of auxiliary category examples (*NoObject*, *ExistingObject*, *OutsideRoom*) in the dataset used to train the `CategoryLocation` network. Figure 3.6 shows typical predicted distributions for each setting. In the top row, we show how all the features are needed to prevent the predicted location distribution for beds to place probability mass in locations that would block the door (upper right of the room). In the bottom row, we show how increasing the percentage of auxiliary category examples in the training data removes spurious probability mass from the distribution for wardrobe locations, leading to a distribution that plausibly concentrates its mass along available walls.

Sampling.

To sample a new category and location at synthesis time, we need to construct a joint distribution $p(c, x, y|\mathcal{S})$ using the values of $p_{\textbf{cat}}(c|\mathcal{S}, x, y)$. To make this problem tractable, we consider candidate locations on a $N \times N$ regular grid defined over the scene. We then need to define a prior distribution $p(x, y|\mathcal{S})$ over grid locations. Since we have already taken into account the probability of objects being present/absent at a particular location through the use of auxiliary categories, we can use a uniform prior $p(x, y|\mathcal{S}) = \frac{1}{NN}$. Our joint distribution is then, by the chain rule:

$$p(c, x, y|\mathcal{S}) = p_{\textbf{cat}}(c|\mathcal{S}, x, y) \cdot \frac{1}{NN}$$

For brevity, we drop the conditioning on the current scene $\mathcal{S}$ for the remainder of this section.

$p(c, x, y)$ is a three dimensional discrete distribution of size $C \times N \times N$, where C is the number of categories. When sampling from large generative models such as this one trained on a high volume of noisy data, it is common practice to suppress noise in the distribution by adjusting the model's *temperature*, i.e. taking $p^{1/\tau}$ for some temperature value τ (see e.g. [3, 46]). We also temper the joint category-location distribution $p(c, x, y)$, though we have found a two-step tempering scheme to perform better. We first temper the spatial probability distributions for each category c:

$$\begin{aligned} p(c) &= \sum_{x=1}^{N} \sum_{y=1}^{N} p(c, x, y) \\ p^{1/\tau_1}(x, y|c) &= \frac{p(c, x, y)^{1/\tau_1}}{\sum_{x=1}^{N} \sum_{y=1}^{N} p(c, x, y)^{1/\tau_1}} \\ f_1(c, x, y) &= p^{1/\tau_1}(x, y|c) \cdot p(c) \end{aligned}$$

Here, $f_1(c, x, y)$ is a density with the same shape as $p(c, x, y)$, but with noise suppressed and high-probability modes highlighted for each category c. We then temper the overall spatial density $f_1(x, y)$:

$$\begin{aligned} f_1(x, y) &= \sum_{c=1}^{C} f_1(c, x, y) \\ f_1(c|x, y) &= f_1(c, x, y)/f_1(x, y) \\ f_2(c, x, y) &= f_1(c|x, y) \cdot f_1(x, y)^{1/\tau_2} \end{aligned}$$

After this step, $f_2(c, x, y)$ has suppressed overall low-probability locations, but without changing the relative probability of categories at any location. We then normalize f_2 to produce the final distribution p^* from which our model samples:

$$p^*(c, x, y) = \frac{f_2(c, x, y)}{\sum_{c=1}^{C} \sum_{x=1}^{N} \sum_{y=1}^{N} f_2(c, x, y)}$$

We set $\tau_1 = 0.25$ and $\tau_2 = 0.4$ for all the results shown in this chapter.

At synthesis time, we use a 32×32 grid (i.e. $N = 32$) to sample a location (x, y) and a category c. To obtain a more fine-tuned location, we then use a 16×16 grid within the previously-sampled grid cell, constrain the category to c, and choose the location which maximizes $p^*(x, y|c)$).

3.3.3 Placing an object instance

Given a location (x, y) and an object category c, our model's final step is to insert an instance of that category at that location. Here, 'instance' refers to a specific 3D model. In our work, we draw these 3D models from the set of models used in the SUNCG dataset.

There can be many possible instances for a given category (for example, there are nearly 300 chair models in SUNCG), not all of which are visually compatible with one another. Thus, we first restrict the set of instances considered to those likely to be compatible with existing objects in the room. SUNCG, having originated from an interior design tool, contains many 3D models drawn from stylistically-consistent *collections* of furniture. We annotated all SUNCG 3D models with their collection. At synthesis time, we allow the insertion of instances that come from the same collection as another object already in the room. We relax this constraint slightly by also allowing instances that co-occur in the same SUNCG scene with another object already in the room. Prior work has developed more sophisticated approaches for measuring the style compatibility of 3D furniture models in terms of their geometry [74] or materials [22]; such methods could also be applied here.

To insert an instance into the room, our model must also choose an orientation for that instance, where orientation is defined as a single angle θ about the gravity vector. To do this, we use a third and final convolutional network component, `InstanceOrientation`. This component inserts a candidate instance i into the scene at a candidate orientation θ, and then evaluates a network which returns $p_{\textbf{inst}}(\top|\mathcal{S}, i, x, y, \theta)$, the probability that inserting instance i at (x, y, θ) is a valid addition to the scene $\mathcal{S}$. `InstanceOrientation` augments the top-down scene view $\mathcal{V}(\mathcal{S})$ with a mask for the geometry of the inserted instance,

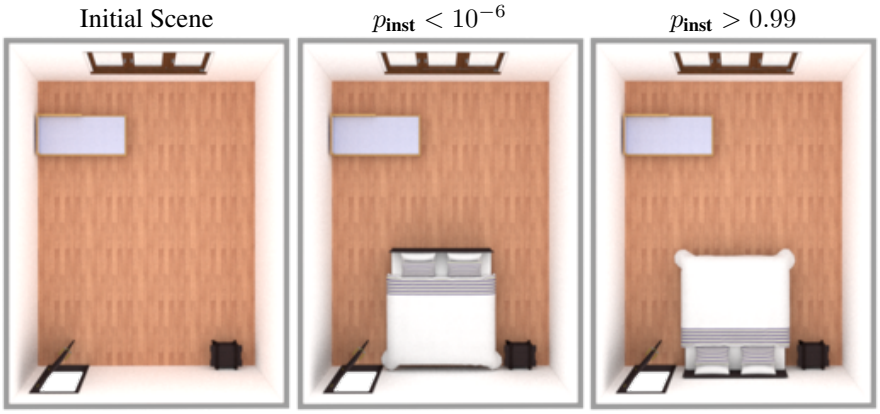


Figure 3.7: Using the `InstanceOrientation` network to predict the validity of inserting model instances at different orientations. Inserting the double bed at the correct orientation with respect to the wall has probability near 1, whereas the opposite orientation is clearly recognized as being incorrect with probability near 0.

$\mathcal{M}_{\textbf{geo}}(i, x, y, \theta)$. It computes the probability of the insertion as

$$p_{\textbf{inst}}(\top|\mathcal{S}, i, x, y, \theta) = \sigma(\text{MLP}(\text{CNN}([\mathcal{V}(\mathcal{S}), \mathcal{M}_{\textbf{geo}}(i, x, y, \theta)])))$$

Figure 3.7 shows an example of $p_{\textbf{inst}}$ discriminating between a valid and an invalid insertion.

We train `InstanceOrientation` by taking rooms from our dataset, again removing a random set of objects from each room, and then selecting one removed object to be re-inserted. We generate 50% positive training examples by inserting the object at its original orientation, and 50% negative training examples by inserting the object at a different orientation. For these negative example orientations, we quantize $[0, 2\pi]$ into 16 discrete orientations with the object's original orientation at 0, and we select from among the 15 other orientations. Training uses the standard binary cross entropy loss.

At synthesis time, our model tries to insert all of the allowed instances at each of 16 discrete orientations and chooses the collision-free insertion with highest probability. If there is no collision free insertion or no insertion with probability higher than 50%, our model resamples a different (x, y, c) tuple from `CategoryLocation`. We automatically reject any synthesized room that uses more than 20 such resampling steps. In our experiments, this was $\sim 20\%$ of synthesized bedrooms, $\sim 15\%$ of offices, and $< 1\%$ of living rooms. Rooms that are not rejected use on average less than three resampling steps.

3.3.4 Details and timings

We implement our neural networks in PyTorch [88]. All the convolutional networks use the same architecture, the Resnet-101 architecture [47] (modified to use 512×512 images as input). The set of features $f_{\mathbf{cnn}}$ output by these networks has size 2048. The MLP networks contain three fully-connected layers with input sizes $2048 + C$, 2048, and 1024, respectively. We use ReLU activation and batch normalization between these layers.

We train our networks on NVIDIA GeForce GTX 1080 Ti GPUs, using the Adam optimizer [61]. Training takes 6 hours for `Continue?` and 2 days for `CategoryLocation` and `InstanceOrientation`. `Continue?` and `InstanceOrientation` can both be interpreted as binary classifiers, so we can evaluate their classification accuracy: `Continue?` reaches $\sim 85\%$ validation accuracy on average across all room types, and `InstanceOrientation` reaches $\sim 94\%$.

At synthesis time, evaluating `CategoryLocation` for a 32×32 grid of scene locations takes 25 seconds, and placing an object instance takes on average 20 seconds. This lead to an average overall room synthesis time of 4 minutes. In Section 3.5, we discuss possible approaches to reduce this computation time.

3.4 Results and Evaluation

In this section, we evaluate our model's ability to generate plausible rooms with characteristics similar to the SUNCG training scenes. To do this, we compare scenes synthesized by our model to several baselines using perceptual studies on Amazon Mechanical Turk. All of these comparisons use the same study design, which extends that of prior work on evaluating automatic image colorization methods [150]. Study participants were given a series of forced-choice image comparison tasks. Each task displayed a pair of images, one depicting a scene synthesized by our model and the other from a baseline source. Both scenes used the same room geometry. The order of images within each pair was randomized. The objects in each scene image were colored according to their category, so that participants would make judgments based on the types of objects present and their arrangement, and not based on extraneous factors such as object materials. The task instructed participants to choose the image from the pair which they think depicts a more plausible arrangement of objects in the room. Each participant performed 55 comparison tasks. One out of each 11 tasks was a 'vigilance test': a comparison with an obviously wrong answer (specifically, one image depicted a randomized, jumbled arrangement of random objects). For each study on each room type, we collected responses from 10 AMT

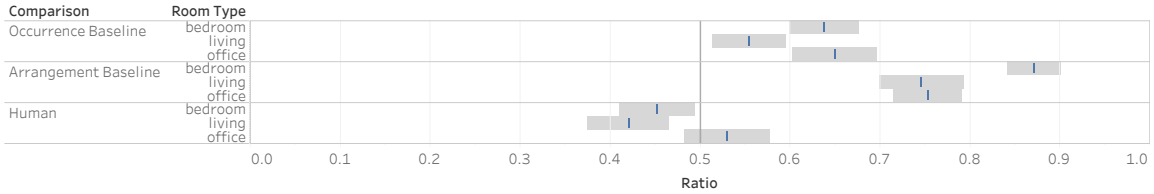


Figure 3.8: Results of Mechanical Turk forced-choice perceptual studies comparing scenes synthesized by our method to those from several baselines. Blue lines show the percentage of time that our method was chosen, with gray bars showing 95% confidence intervals computed by bootstrap [28]. Over all room types, scenes generated by our method are significantly preferred to those generated by first sampling a set of objects and then arranging them using our model (*Occurrence Baseline*). When arranging a fixed set of objects, our model is also significantly preferred over a model based on pairwise object relationship statistics (*Arrangement Baseline*). Compared to human-created scenes from our test dataset, our generated scenes are equally preferred for office scenes, and slightly less preferred for bedroom and living room scenes.

Master workers (workers AMT identifies as consistently high-performing) and discarded all responses from workers who did not achieve 100% accuracy on the vigilance tests.

Comparison against independently selecting scene objects

Most prior scene synthesis methods first generate a set of objects for the scene, then arrange those objects. In contrast, our method adds each object based on the existence and arrangement of all objects added before it. In theory, our method can more reliably produce higher-quality scenes, since it is sensitive to the size and shape of the room and what objects can plausibly be used in it, as opposed to independently generating a set of objects that is plausible 'on average.' We conduct an experiment to determine whether this advantage is visible in practice.

As a baseline method for independently generating a set of object categories, we use a Neural Autoregressive Distribution Estimator (NADE) [118]. A NADE is a learnable model for probability distributions over a sequence of variables, where the distribution over variable i is a function of the values of variables 1 through $i-1$. We model distribution over the number of instances of each object category occurring in a scene, which can be represented as a sequence of categorical variables $(x_1, x_2, \ldots)$, with one variable for each category of object that may occur. If x_i is a variable with D_i possible count values (including zero), then its probability

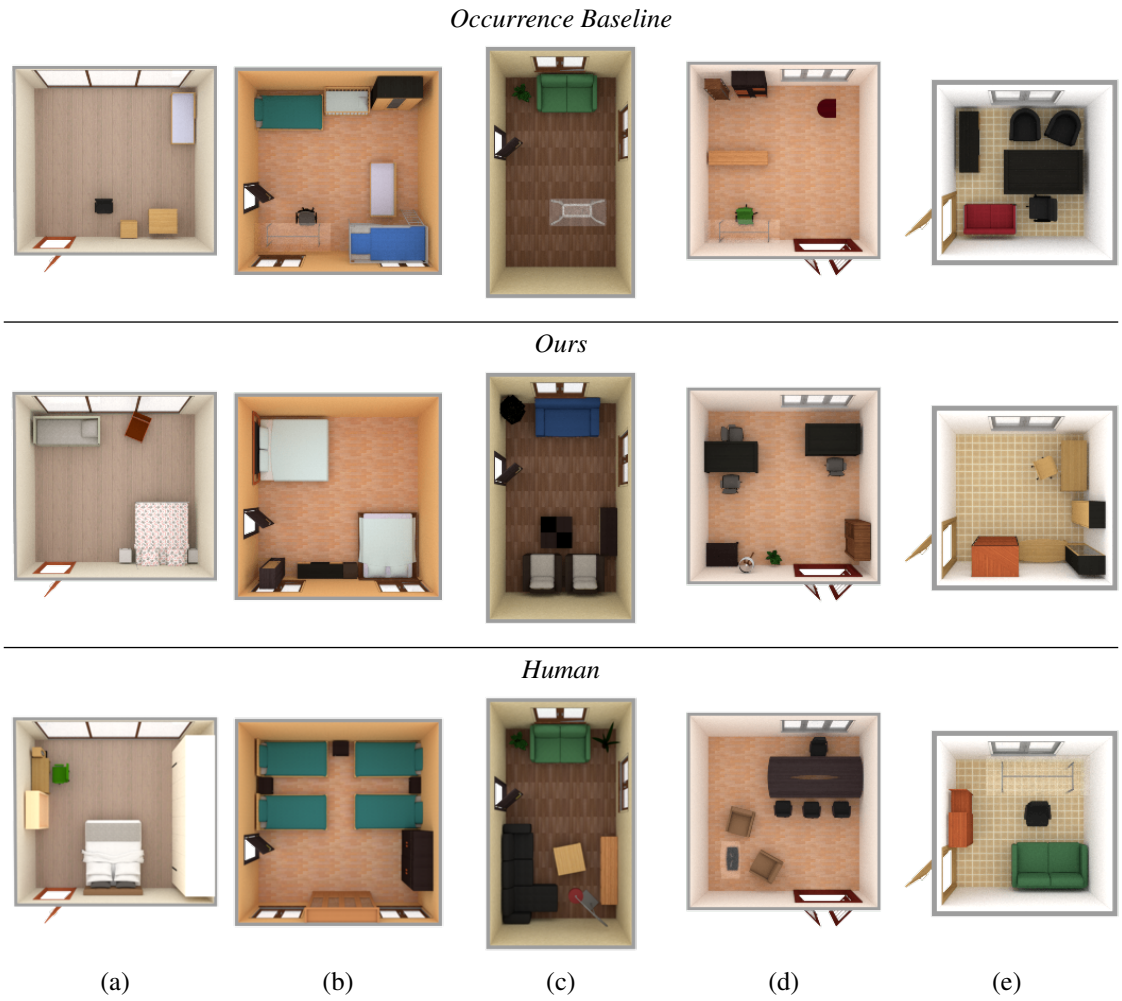


Figure 3.9: Comparison of outputs using baseline occurrence model (top) and our full model (middle), with original human-designed scenes (bottom). Columns (a), (c) and (d) show examples where the baseline model results in too sparsely populated rooms. In column (b), the baseline selects different bed types for the four beds. In column (e) the baseline selects objects that are too large for the space. In contrast, our model is informed by the current arrangement and more plausibly populates the space of the room with consistent object types.

distribution is a categorical distribution computed as follows:

$$p(x_i|\mathbf{x}_{<i}) = \text{softmax}(\mathbf{V}_i\mathbf{h}_i + \mathbf{b}_i)$$

$$\mathbf{h}_i = \text{sigmoid}(\mathbf{a}_i)$$

$$\mathbf{a}_i = x_{i-1}\mathbf{w}_i + \mathbf{a}_{i-1} \text{ , where } \mathbf{a}_1 = \mathbf{c}$$

where $\mathbf{V}_i \in \mathbb{R}^{D_i \times H}$, $\mathbf{b}_i \in \mathbb{R}^{D_i}$, $\mathbf{w}_i \in \mathbb{R}^H$, $\mathbf{c} \in \mathbb{R}^H$ are the learnable parameters of the NADE, with H

being the NADE's hidden layer size. We use this model as a stand-in for Bayesian networks, topic models, and other approaches that have been used to model the distribution of objects in a scene. We train the NADE with the same set of training scenes used to train our model.

In our experiment, we synthesize scenes using the NADE-based object occurrence model as follows: We first use the NADE to sample occurrence counts for all possible object categories in the scene. We then select object instances for these categories and arrange them using our model—this isolates the specific object occurrence behavior we are interested in comparing. To do this, we rejection sample from our model until it chooses to add one of the categories for which the NADE sampled a nonzero count, and we repeat this process until all of the sampled category counts have been satisfied.

In a perceptual study, participants preferred scenes generated by our model over those generated by the baseline across all types of rooms (Figure 3.8, row *Occurrence Baseline*). Figure 3.9 shows some representative scenes generated by each method, along with a human-created scene from the dataset that uses the same room geometry. In many cases, the baseline occurrence model samples a set of objects for the room that is plausible, but too sparse for the particular room geometry (columns (a), (c), and (d)). The opposite phenomenon can also occur, with the baseline model sampling a set of objects that causes the room to be too crowded (column (e) Top, the sofa is forced into a position which blocks the door). Our model, by basing its decision of whether to add more objects and which to add on the current state of the scene, generally avoids these issues.

Comparison against arrangement using pairwise relationship priors

We also conducted an experiment to see whether our model suggests better arrangements of objects than models based on simpler, pairwise relationship statistics. In our experiment, we compare the ability of the two approaches to arrange the same, fixed set of objects—this isolates the specific object arrangement behavior we are interested in evaluating. For the fixed set of objects, we use the objects from a scene in our dataset which was not used for training. This also allows us to compare against the arrangement for these objects chosen by the original human author of the scene. This is a challenging task for any method like ours which attempts to greedily add objects one a time into the scene: many scenes in the dataset are tightly packed and precisely arranged, making it difficult to produce an alternative plausible arrangement. We are interested in whether our priors allow our model more frequently to find a reasonable re-arrangement of the scene objects.

As a baseline method for arranging objects, we use a set of learned pairwise priors based on prior work. In constructing this baseline, our goal was to select a set of commonly used elements in prior approaches

Figure 3.10: Comparison of outputs using baseline arrangement model (top) and our full model (middle), with original human-designed scenes (bottom). In columns (a) and (b), the baseline arranges objects packed too close together and not fully making use of space against the room walls. In column (c), the space in front of the L-shaped couch is taken up by a chair instead of the glass coffee table. In column (d), both chairs are placed close to one of the tables, while in column (e), the blue couch is too close to the desk. In contrast, our model generates arrangements that more plausibly make use of free space.

for object layout optimization. We learn priors for the pairwise relative position and orientation of objects, and distance and angle to the closest wall for each object category. Variants of these priors are used by much prior work in 3D interior arrangement [149, 78, 34, 60]. Though priors on other properties such as symmetry [60], conversation [78], human activities [35, 38], and door-to-door navigability [149] have been used in optimizing object arrangement, our goal is to contrast our method against a representative purely data-driven approach using only learned pairwise object placement priors. We represent these pairwise priors as mixtures of Gaussian distributions encoding the offset between a pair of objects, and a discrete distribution

encoding the angle in the 2D plane between the front orientations of the two objects (including walls). During synthesis, like our method, the baseline adds objects one at a time, in order of decreasing physical size. It evaluates the probability of many randomly-sampled candidate placements of the object using the pairwise priors of that object with respect to all existing objects in the scene, and it chooses the highest-probability position and orientation. Appendix A.2 provides more details.

In the perceptual study, participants preferred re-arrangements generated by our method to those generated by the baseline (Figure 3.8, row *Arrangement Baseline*). Figure 3.10 shows representative scenes compared by participants in this study. While both methods struggle to re-arrange especially tightly-packed scenes, ours performs better in most cases. Columns (a) and (b) show two bedrooms that our model manages to re-arrange plausibly, while the baseline model resorts to implausible, non-navigable layouts to pack everything into the available space. Column (e) shows re-arrangement of a crowded office; our model places the sofa against a wall, whereas the baseline places it in an unusable configuration close to the desk. In column (c), the L-shape of the room makes it difficult to arrange the sofa, table, and chair around each other as is typical. Our method packs the table too tightly into the sofa's concavity, but this is preferable to putting the chair there instead, as the baseline does. Finally, column (d) illustrates a known failure mode of pairwise priors: conflict between multiple strong pairings. In this case, both chairs cluster around one of the two desks in the room. Our method tends to handle this case better, as this result and the other synthesized offices in Figure 1.1 illustrate. Some of the baseline's shortcomings could be remedied via explicit human-factors penalty terms, such as those mentioned in the previous paragraph. Such terms could also improve our method, and we find it interesting and encouraging that our model performs as well as it does given that it is purely data-driven.

Comparison against human-created scenes

We then compare scenes synthesized by our model with held-out test scenes from our dataset. Figure 1.1 shows examples of such scenes for each room type. In the perceptual study, participants showed no preference between the two types of scenes for offices, and they slightly preferred the human-created scenes for bedrooms and living rooms. While our model generates plausible scenes on average, it does exhibit a few failure modes, some typical instances of which are shown in Figure 3.13. In the top-left living room scene, the model has placed plausible objects in plausible locations, but it did not place any seating objects such as sofas or chairs. This is partly a product of some training set living rooms having this characteristic, and partly that our model is more sensitive to local scene plausibility than global plausibility. The top-right bedroom has too many nightstands next to bed, as beds are a strong cue for nightstands, and the presence of one does

Figure 3.11: Synthesizing multiple scenes in the same room. *Bedroom:* three different orientations of the bed and different wall-adjacent objects. *Living Room:* one room with a TV stand and three without, each with distinct sofa/table layouts. *Office:* synthesized rooms accommodate different numbers of occupants and place sofas and bookshelves where space is available.

not always fully suppress the probability of adding another in the same or nearby location. In the bottom-left office scene, `CategoryLocation` sampled a reasonable centroid location for a wardrobe cabinet, but the model inserted by `InstanceOrientation` is very long and partially blocks the door. Finally, the lower-right bedroom scene contains two beds that are plausibly arranged when considered independently but whose geometry does not leave room between them to traverse the room. Improving global consistency and enforcing a stronger coupling between object location and geometry are important avenues for future work, and challenging problems for scene synthesis in general.

Figure 3.12: Examining our model's generalization capability. *Top row:* Bedrooms synthesized by a model trained on 320 rooms. *Bottom row:* For each synthesized room, we show its nearest neighbor in the training set. *From left to right:* adding a second bed to a common layout in the dataset; different desk/cabinet placement, given a similar bed position; generalizing to L-shaped rooms, which do not occur in the training set.

Scene variety and generalization

Finally, we examine our model's ability to generate a variety of scenes and generalize beyond its training dataset. Figure 3.11 shows multiple scenes synthesized in the same input room. Our model explores different possible positions for key objects (such as beds and sofas), chooses different combinations of objects, and generates rooms with varying functionality (e.g. living rooms with/without televisions, offices accommodating different numbers of people). Figure 3.12 shows four bedrooms synthesized by a model trained on a small dataset of 320 rooms. We also show the nearest neighbor of each room in the training set, using a distance function adapted from prior work [48]. Even with this smaller training set, our model generalizes and generates reasonable layouts that differ from rooms in the training set. This training set also contains no L-shaped rooms, yet our model can still generate layouts for them that respect the room shape.

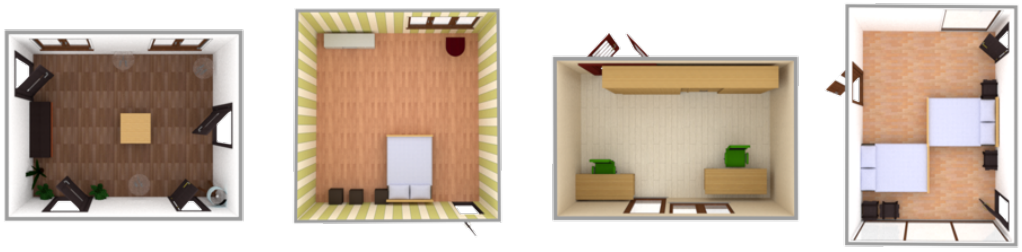


Figure 3.13: Typical failure cases of our model. From left to right: the living room contains no seats (global inconsistency), the bedroom has too many nightstands (local/global inconsistency), the wardrobe cabinet blocks the door (conflict between object location and geometry), the two beds don't leave enough room to walk between (global inconsistency, conflict between object location and geometry).

Figure 3.14: Room types that our model currently does not handle well. *Left:* A human-designed kitchen and dining room. *Right:* Our approach struggles with generating carefully coordinated functional groups of objects, such as the contiguous placement of separate kitchen countertop sections and the symmetric arrangement of chairs around dining tables.

3.5 Chapter Summary

This chapter presented Deep Synth, the first deep convolutional neural network-based system for generating object arrangements of entire rooms, given only the room wall architecture as input. Our multi-channel top-down view representation encodes the context of an entire room and enables the application of convolutional networks to the domain of 3D scene synthesis. We demonstrated that our system learns to iteratively condition object selection and placement on the global state of the room during synthesis. Finally, we evaluated the plausibility of rooms generated using our approach against representative object selection and arrangement baselines based on prior work.

Our system has several key limitations. We represent scenes as flat collections of objects. However, indoor scenes exhibit hierarchy, with sets of objects forming 'functional groups' (e.g. *bed-and-nightstands*, *table-and-chairs*). Such groups also often exhibit symmetries, e.g. chairs being symmetrically arranged

around a table. Our model currently struggles with such carefully-coordinated groupings of many objects (Figure 3.14). Explicitly incorporating hierarchy and symmetry into our model could help address these problems, along with allowing us to tackle larger-scale scenes, such as classrooms, restaurants, and corporate offices (which the SUNCG dataset contains). We attempt to address this limitation in Chapter 5.

As mentioned in Section 3.3.4, sampling a scene from our model takes several minutes on average. Much of this computation time is spent in evaluating the `CategoryLocation` network for many candidate locations in the scene. This phase could be sped up if the computations for multiple locations could be shared. One possible approach would be to draw inspiration from fast region proposal networks for object detection, which use a common feature map for the whole image when evaluating candidate object regions [97]. Or, we could design the `CategoryLocation` network as an image-to-image translation function, predicting category probabilities across all pixels of the output image in one forward pass [52]. Predicting category probabilities across an entire room at training could also facilitate better global consistency, reducing the need for `CategoryLocation`'s global loss $\mathcal{L}_{\textbf{global}}$ and for tempering during sampling. In Chapter 4, we incorporate some of these ideas to significantly speed up the process of sampling a scene.

The model as described considers only floor-supported (i.e. 'first-tier') objects. It would be straightforward to incorporate 'second-tier' objects (i.e. those supported by other objects) using a second pass of our model, after all first-tier objects have been placed. We adopt this "two-pass" strategy in Chapter 4. In Chapter 5, we use a graph-based scene representation that explicitly defines "support" relationships between first-tier and second-tier objects. It is also possible to include wall-mounted objects such as clocks and paintings by using multiple orthographic views as input to our networks, i.e. a top-down view along with a wall-facing view. We leave that for future work.

More broadly, our model makes no major assumptions specific to the domain of interior room design. We believe that deep convolutional priors extracted from multi-channel view representations such as the ones we have presented open up new possibilities for many application domains that involve reasoning over 3D scene structure.

Chapter 4

Fast and Flexible Indoor Scene Synthesis via Deep Convolutional Generative Models

As mentioned in the previous chapter, our first attempt in using deep convolutional neural networks (CNNs) to perform scene synthesis, while promising, suffers from several limitations. It reasons locally about object placements and can struggle to globally coordinate an entire scene (e.g. failing to put a sofa into a living room scene). It does not model the size of objects, leading to problems with inappropriate object selection (e.g. an implausibly-long wardrobe which blocks a doorway). Finally, and most critically, it is extremely slow, requiring minutes to synthesize a scene due to its use of hundreds of deep CNN evaluations per scene.

We believe that image-based synthesis of scenes is promising because of the ability to perform precise, pixel-level spatial reasoning, as well as the potential to leverage existing sophisticated machinery developed for image understanding with deep CNNs. In this chapter, we present a new image-based scene synthesis pipeline, based on deep convolutional generative models, that overcomes the issues of our previous work. This new method also generates scenes by iteratively adding objects. The key difference, however, is that it factorizes the step of adding each object into a different sequence of decisions. In the previous chapter, an object is inserted with three steps: 1. whether to stop; 2. category and location; 3. model instance and its orientation. The second step requires sampling an attention mask across the entire room, leading to hundreds, if not thousands, of inferences. The third step requires trying all reasonable CAD models, each

in 16 orientations, and is also prohibitively expense. Instead, in this chapter, we propose a new factorization of four steps: 1. category, where a STOP category is added to indicate completion; 2. location, with a new approach based on Fully Convolutional Networks (FCN); 3. orientation 4; dimension, which is used to sample CAD models of matching size. Each of this step requires only a single forward inference, leading to a pipeline that is two orders of magnitude faster than our prior work, requiring on averange under 2 seconds to synthesize a scene. It also allows the new method to 1. to reason globally about which objects to add, and 2. to model the spatial extent of objects to be added, in addition to their location and orientation.

We evaluate our method by using it to generate synthetic bedrooms, living rooms, offices, and bathrooms (Figure 1.2). We also show how, with almost no modification to the pipeline, our method can synthesize multiple automatic completions of partial scenes using the same fast generative procedure. We compare our method to our prior work, another state-of-the art deep generative model based on scene hierarchy, and scenes created by humans, in several quantitative experiments and a perceptual study. Our method performs as well or better than these prior techniques.

4.1 Model

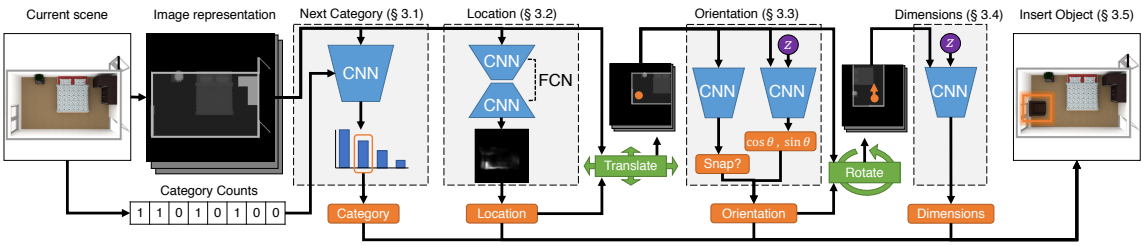


Figure 4.1: Overview of our automatic object-insertion pipeline. We extract a top-down-image-based representation of the scene, which is fed to four decision modules: which category of object to add (if any), the location, orientation, and dimensions of the object.

Our goal is to build a deep generative model of scenes that leverages precise image-based reasoning, is fast, and can flexibly generate a variety of plausible object arrangements. To maximize flexibility, we use a sequential generative model which iteratively inserts one object at a time until completion. In addition to generating complete scenes from an empty room, this paradigm naturally supports partial scene completion by simply initializing the process with a partially-populated scene. Figure 4.1 shows an overview of our pipeline. It first extracts a top-down, floor-plan image representation of the input scene, as done in the previous chapter. Then, it feeds this representation to a sequence of four decision modules to determine

how to select and add objects into the scene. These modules decide which category of object to add to the scene, if any (Section 4.1.1), where that object should be located (Section 4.1.2), what direction it should face (Section 4.1.3), and its physical dimensions (Section 4.1.4). This is a different factorization than in prior work, which we will show leads to both faster synthesis and higher-quality results. The rest of this section describes the pipeline at a high level; precise architectural details can be found in Appendix A.3, and the source code for our system is available at `https://github.com/brownvc/fast-synth`.

4.1.1 Next Object Category

The goal of our pipeline's first module is, given a top down scene image representation, to predict the category of an object to add to the scene. The module needs to reason about what objects are already present, how many, and the available space in the room. To allow the model to also decide when to stop, we augment the category set with an extra "<STOP>" category. The module uses a Resnet18 [47] to encode the scene image. It also extract the counts of all categories of objects in the scene (i.e. a "bag of categories" representation), as in the previous chapter, and encodes this with a fully-connected network. Finally, the model concatenates these two encodings and feeds them through another fully-connected network to output a probability distribution over categories. At test time, the module samples from the predicted distribution to select the next category.

Figure 4.2 shows some example partial scenes and the most probable next categories that our model predicts for them. Starting with an empty scene, the next-category distribution is dominated by one or two large, frequently-occurring objects (e.g. beds and wardrobes, for bedroom scenes). The probability of other categories increases as the scene begins to fill, until the scene becomes sufficiently populated and the "<STOP>" category begins to dominate.

In this previous chapter, we predicted category and location jointly. This lead to the drawback that objects which are very likely to occur in a location can be repeatedly (i.e. erroneously) sampled, e.g. placing multiple nightstands to the left of a bed. In contrast, the new category prediction module reasons about the scene globally and thus avoid this problem.

4.1.2 Object Location

In the next module, our model takes the input scene and predicted category to determine where in the scene an instance of that category should be placed. We treat this problem as an image-to-image translation problem:

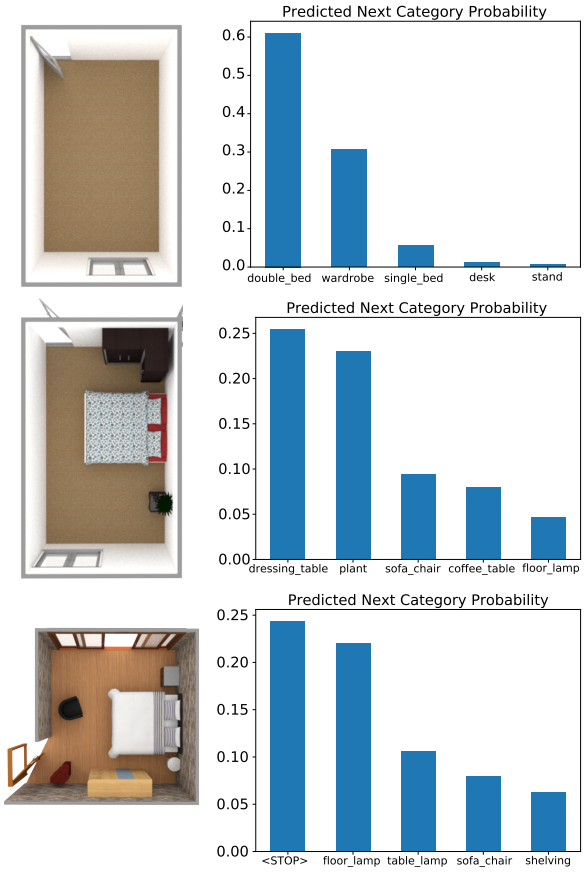


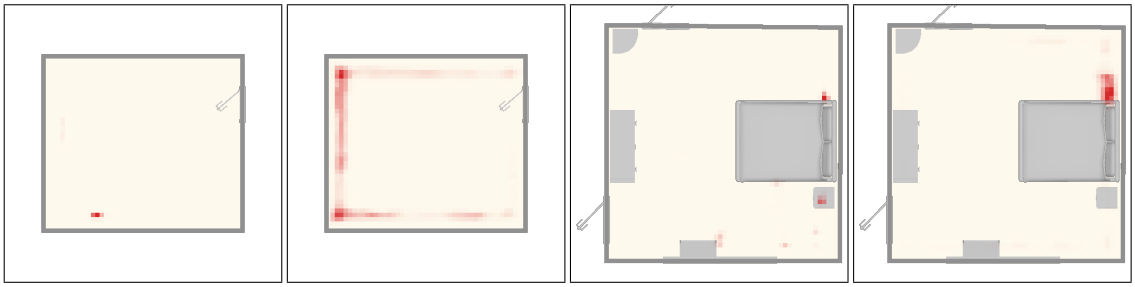
Figure 4.2: Distributions over the next category of object to add to the scene, as predicted by our model. Empty scenes are dominated by one or two large, frequent object types (*top*), partially populated scenes have a range of possibilities (*middle*), and very full scenes are likely to stop adding objects (*bottom*).

given the input top-down scene image, output a 'heatmap' image containing the probability per pixel of an object occurring there. This representation is advantageous because it can be treated as a (potentially highly multimodal) 2D discrete distribution, which we can sample to produce a new location. This pixelwise discrete distribution is similar to that in the previous chapter, except that in the previous chapter, we assembled the distribution pixel-by-pixel, invoking a deep convolutional network once per pixel of the scene. In contrast, our new module uses a single forward pass through a fully-convolutional encoder-decoder network (FCN) to predict the entire distribution at once.

This module uses a Resnet34 encoder followed by an up-convolutional decoder. The decoder outputs a $64 \times 64 \times |C|$ image, where $|C|$ is the number of categories. The module then slices out the channel corresponding to the category of interest and treats it as a 2D probability distribution by renormalizing it. We also experimented with using separate FCNs per category that predict a $64 \times 64 \times 1$ probability density image

Double Bed *Wardrobe* *Nightstand*

Figure 4.3: Probability densities for the locations of different object types predicted by our fully-convolutional network module.



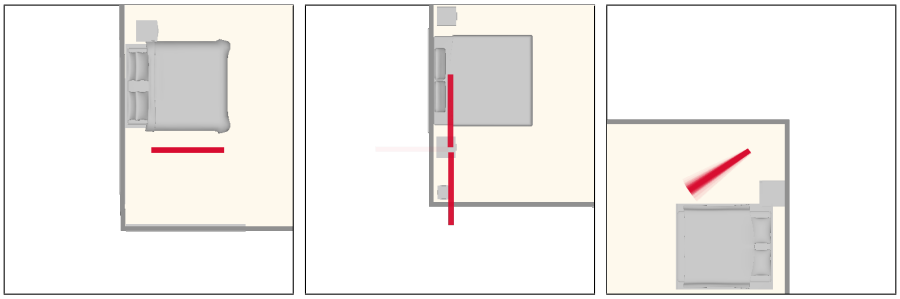
(a) (b) (c) (d)

Figure 4.4: Probability distributions for nightstands, without (*(a)* & *(c)*) and with (*(b)* & *(d)*) regularization.

but found it not to work as well. We suspect that training the same network to predict all categories provides the network with more context about different locations, e.g. instead of just learning that it should not predict a wardrobe at a location, it can also learn that this is because a nightstand is more likely to appear there. Before renormalization, the module zeros out any probability mass that falls outside the bounds of the room. When predicting locations for second-tier categories (e.g. table lamps), it also zeros out probability mass that falls on top of an object that was not observed as a supporting surface for that category in the dataset. At test time, we sample from a tempered version of this discrete distribution (we use temperature $\tau = 0.8$ for all experiments in this chapter).

Figure 4.3 shows examples of predicted location distributions for different scenes. The predicted distributions for bed and wardrobe avoid placing probability mass in locations which would block the doors. The distribution for nightstand is bimodal, with each mode tightly concentrated around the head of the bed.

To train the network, we use pixel-wise cross entropy loss. As in the previous chapter, we augment the category set with a category for "empty space," which allows the network to reason about where objects should *not* be, in addition to where they should. Empty-space pixels are weighted 10 times less heavily than occupied pixels in the training loss computation. As the ground truth label for each training example is



Nightstand *Table Lamp* *Armchair*

Figure 4.5: High-probability object orientations sampled by our CVAE orientation predictor (visualized as a density plot of front-facing vectors). Objects typically either snap to one orientation (*left*) or multiple orientation modes (*middle*), or have a range of values clustered around a single mode (*right*).

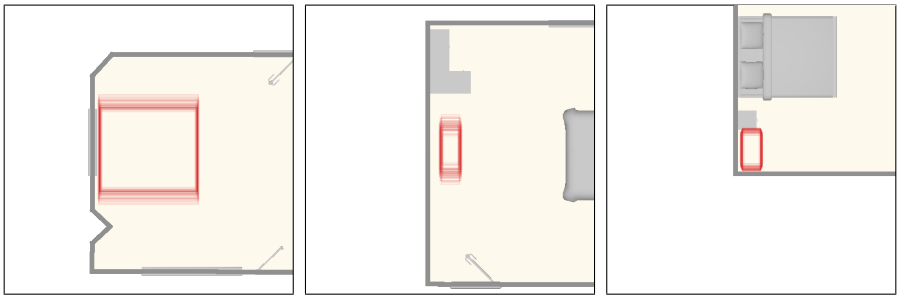
a single location instead of a distribution, our model has the potential to overfit to that exact location. This is shown in Figures 4.4a & 4.4c, where the predicted distribution collapses to single-point locations. In the second case, the network likely tries to match the input room to several memorized ones, none of which makes sense. To deal with this problem, we handicap the capacity of the network by applying L2 regularization and dropout, forcing it to learn a latent space where structurally similar scenes are close together. This results in averaged output locations, i.e. a continuous distribution of locations (Figures 4.4b & 4.4d).

Before moving on to the next module, our system translates the input scene image so that it is centered about the predicted location. This makes the subsequent modules translation-invariant.

4.1.3 Object Orientation

Given a translated top-down scene image and object category, the orientation module predicts what direction an object of that category should face if placed at the center of the image. We assume each category has a canonical front-facing direction. Rather than predict the angle of rotation θ, which is circular, we instead predict the front direction vector, i.e. $[\cos\theta, \sin\theta]$. This must be a normalized vector, i.e. the magnitude of $\sin\theta$ must be $\sqrt{1-\cos^2\theta}$. Thus, our module predicts $\cos\theta$ along with a Boolean value giving the sign of $\sin\theta$. Here, we found using separate network weights per category to be most effective.

The set of possible orientations has the potential to be multimodal: for instance, a bed in the corner of a room may be backed up against either wall of the corner. To allow our module to model this behavior, we implement it with a conditional variational autoencoder (CVAE) [113]. Specifically, we use a CNN to encode the input scene, which we then concatenate with a latent code z sampled from a multivariate unit normal distribution, and then feed to a fully-connected decoder to produce $\cos\theta$ and the sign of $\sin\theta$. At



Double Bed *TV Stand* *Ottoman*

Figure 4.6: High-probability object dimensions sampled by our CVAE-GAN dimension predictor (visualized as a density plot of bounding boxes). Objects in more constrained locations have lower-variance size distributions (*right*).

training time, we use the standard CVAE loss formulation (i.e. with an extra encoder network) to learn an approximate posterior distribution over latent codes).

Since interior scenes are frequently enclosed by rectilinear architecture, objects in them are often precisely aligned to cardinal directions. A CVAE, however, being a probabilistic model, samples noisy directions. To allow our module to produce precise alignments when appropriate, this module includes a second CNN which takes the input scene and predicts whether the object to be inserted should have its predicted orientation "snapped" to the nearest of the four cardinal directions.

Figure 4.5 shows examples of predicted orientation distributions for different input scenes. The nightstand snaps to a single orientation, being highly constrained by its relations to the bed and wall. Table lamps are often symmetric, which leads to a predicted orientation distribution with multiple modes. An armchair to be placed in the corner of a room is most naturally oriented diagonally with respect to the corner, but some variability is possible.

Before moving on to the next module, our system rotates the input scene image by the predicted angle of rotation. This transforms the image into the local coordinate frame of the object category to be inserted, making subsequent modules rotation-invariant (in addition to already being translation-invariant).

4.1.4 Object Dimensions

Given a scene image transformed into the local coordinate frame of a particular object category, the dimensions module predicts the spatial extent of the object. That is, it predicts an object-space bounding box for the object to be inserted. This is also a multimodal problem, even more so than orientation (e.g. many wardrobes of varying lengths can fit against the same wall). Again, we use a CVAE for this: a CNN encodes

the scene, concatenates it with z, and then uses a fully-connected decoder to produce the $[x, y]$ dimensions of the bounding box.

The human eye is very sensitive to errors in size, e.g. an object that is too large and thus penetrates the wall next to it. To help fine-tune the prediction results, we also include an adversarial loss term in the CVAE training. This loss uses a convolutional discriminator which takes the input scene concatenated channel-wise with the signed distance field (SDF) of the predicted bounding box. As with the orientation module, this module also uses separate network weights per category.

Figure 4.6 visualizes predicted size distributions for different object placement scenarios. The predicted distributions capture the range of possible sizes for different object categories, e.g. TV stands can have highly variable length. However, in a situation such as Figure 4.6 Right, where an ottoman is to be placed between the nightstand and the wall, the predicted distribution is lower-variance due to this highly constrained location.

4.1.5 Object Insertion

To choose a specific 3D model to insert given the predicted category, location, orientation, and size, we perform a nearest neighbor search through our dataset to find 3D models that closely fit the predicted object dimensions. When multiple likely candidate models exist, we favor ones that have frequently co-occurred in the dataset with other objects already in the room, as this slightly improves the visual style of the generated rooms (though it is far from a general solution to the problem of style-aware scene synthesis). Occasionally, the inserted object collides with existing objects in the room, or, for second-tier objects, overhangs too much over its supporting surface. In such scenarios, we choose another object of the same category. In very rare situations (less than 1%), no possible insertions exist. If this occurs, we resample a different category from the predicted category distribution and try again.

4.2 Data & Training

We train our model using the SUNCG dataset, a collection of over forty thousand scenes designed by users of an online interior design tool [114]. In this chapter, we focus our experiments on four common room types: bedrooms, living rooms, bathrooms, and offices. We extract rooms of these types from SUNCG, performing pre-processing to filter out uncommon object types, mislabeled rooms, etc. After pre-processing, we obtained 6300 bedrooms (with 40 object categories), 1400 living rooms (35 categories), 6800 bathrooms (22 categories), and 1200 offices (36 categories). Further details about our dataset and pre-processing procedures

can be found in Appendix A.4.

To generate training data for all of our modules, we follow the same general procedure: take a scene from our dataset, remove some subset of objects from it, and task the module with predicting the 'next' object to be added (i.e. one of the removed objects). This process requires an ordering of the objects in each scene. We infer static support relationships between objects (e.g. lamp supported by table) using simple geometric heuristics, and we guarantee that all supported objects come after their supporting parents in this ordering. We further guarantee that all such supported 'second-tier' objects come after all 'first-tier' objects (i.e. those supported by the floor). For the category prediction module, we further order objects based on their *importance*, which we define to be the average size of a category multiplied by its frequency of occurrence in the dataset. Doing so imposes a stable, canonical ordering on the objects in the scene; without such an ordering, we find that there are too many valid possible categories at each step, and our model struggles to build coherent scenes across multiple object insertions. For all other modules, we use a randomized ordering. Finally, for the location module, the FCN is tasked with predicting not the location of a single next object, but rather the locations of *all* missing objects removed from the training scene whose supporting surface is present in the partial scene.

We train each module in our pipeline separately for different room categories. Empirically, we find that the category module performs best after seeing $\sim 300,000$ training examples, and the location module performs best after $\sim 1,000,000$ examples. As the problems that the orientation and dimension models are solving is more local, their behavior is more stable across different epochs. In practice, with use orientation modules trained with $\sim 2,000,000$ examples and dimension modules trained with $\sim 1,000,000$ examples.

4.3 Results & Evaluation

Complete scene synthesis Figure 1.2 shows examples of complete scenes synthesized by our model, given the initial room geometry. Our model captures multiple possible object arrangement patterns for each room type: bedrooms with desks vs. those with extra seating, living rooms for conversation vs. watching television, etc.

Scene completion Figure 4.7 shows examples of partial scene completion, where our model takes an incomplete scene as input and suggests multiple next objects to fill the scene. Our model samples a variety of different completions for the same starting partial scene. This example also highlights our model's ability to

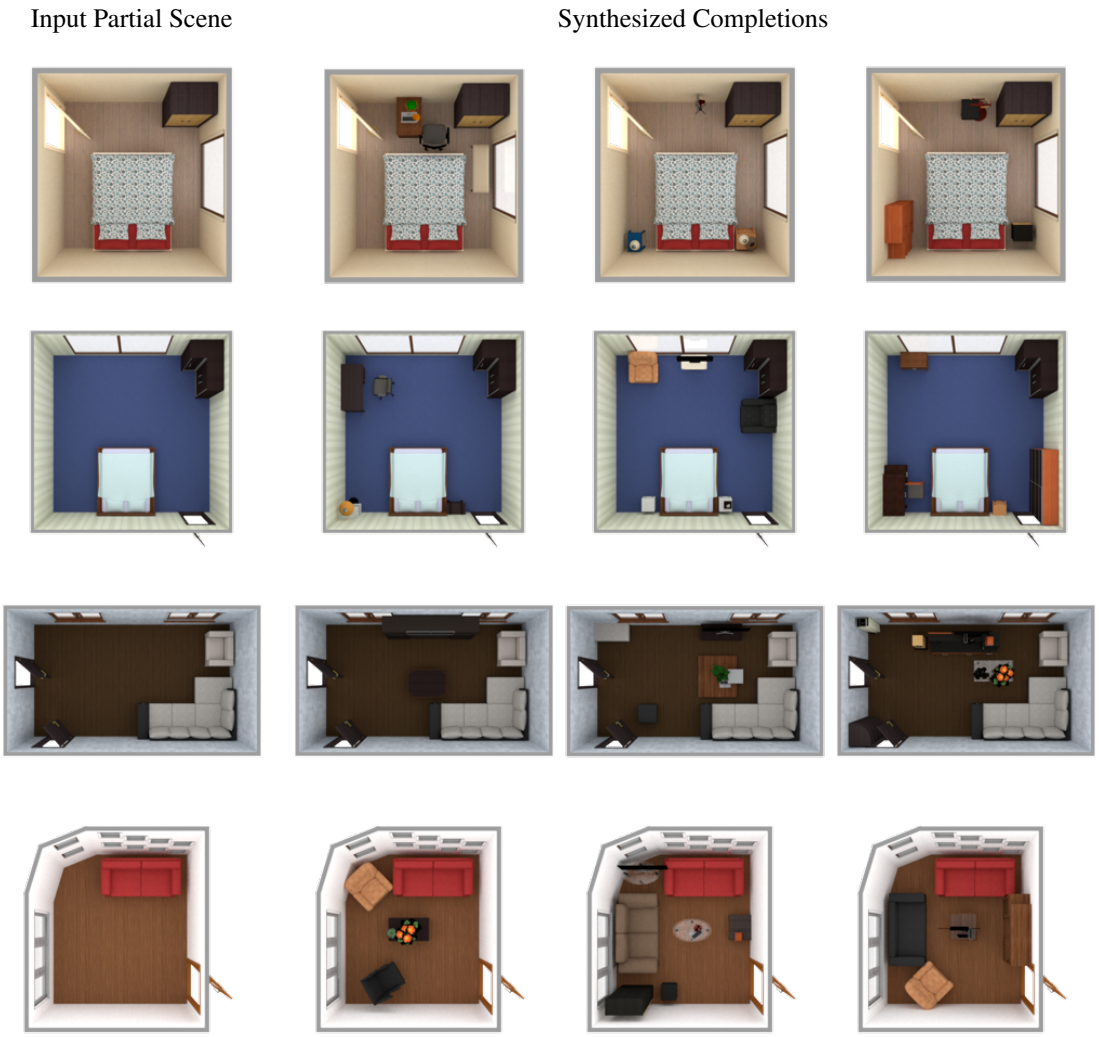


Figure 4.7: Given an input partial scene (*left column*), our method can generate multiple automatic completions of the scene. This requires no modification to the method's sampling procedure, aside from seeding it with a partial scene instead of an empty one.

cope with non-rectangular rooms (bottom row), one of the distinct advantages of precise pixel-level reasoning with image-based models.

Object category distribution For a scene generative model to capture the training data well, a necessary condition is that the distribution of object categories which occurs in its synthesized results should closely resemble that of the training set. To evaluate this, we compute the Kullback-Leibler divergence $D_{\mathrm{KL}}(P_{\mathrm{synth}}||P_{\mathrm{dataset}})$ between the category distribution of synthesized scenes and that of the training set. Note

Method	*Bedroom*	*Living*	*Bathroom*	*Office*
Uniform	0.6202	0.8858	1.3675	0.7219
Deep Synth [125]	0.2017	0.4874	0.2479	0.2138
GRAINS [69]	0.2135	0.3217	—	—
Ours	**0.0095**	**0.0179**	**0.0240**	**0.0436**

Table 4.1: KL divergence between the distribution of object categories in synthesized results vs. training set. Lower is better. *Uniform* is the uniform distribution over object categories.

Method	**Acc**	**Method**	**Acc**
GRAINS [69]	96.56	*No Input Alignment (Orient)*	94.10
Deep Synth [125]	84.69	*No Input Alignment (Dims)*	76.60
Ours	**58.75**	*Joint Category + Location*	81.70
Perturbed (1%)	50.00	*Category from [125]*	89.30
Perturbed (5%)	54.69	*Location from [125]*	83.60
Perturbed (10%)	64.38	*Orient + Dims from [125]*	67.30

Table 4.2: Real vs. synthetic classification accuracy for scenes generated by different methods *(Left)* and our method, modified by changing the design of some of the components or substituting them with similar components from prior works *(Right)*. Lower (closer to 50%) is better.

that we cannot compute a symmetrized Jensen—Shannon divergence because some of the methods we compare against have zero probability for certain categories, making the divergence infinite. Table 4.1 shows the category distribution KL divergence of different methods. Our method generates a category distribution that are more faithful to that of the training set than other approaches.

Scene classification accuracy Looking beyond categories, to evaluate how well the distribution of our generated scenes match that of the training scenes, we train a classifier tasked to distinguish between "real" scenes (from the training set) and "synthetic" scenes (generated by our method). The classifier is a Resnet34 that takes as input the same top-down multi-channel image representation that our model uses. The classifier is trained with 1,600 scenes, half real and half synthetic. We evaluate the classifier performance on 320 held out test scenes.

Table 4.2 shows the performance against different baselines. Compared to previous methods, our results are significantly harder for the classifier to distinguish. In fact, it is marginally *harder* to distinguish our scenes from real training scenes that it is to do so for scenes in which every object is perturbed by a small random amount (standard deviation of 10% of the object's bounding box dimensions).

Effectiveness of our design choices We use the same classification setup to investigate the effectiveness of our individual design choices. As Table 4.2 suggests, swapping out our model components for those in

Figure 4.8: Correcting failure cases from [125], Fig 14. *(Left)* Our model does not omit sofas for seating. *(Right)* Our model chooses a cabinet that does not block the door.

Method	**Avg. Time** (s)
Deep Synth [125]	$\sim$ 240
GRAINS [69]	0.1027
Ours	1.858

Table 4.3: Average time in seconds to generate a single scene for different methods. Lower is better.

the previous chapter [125], omitting input alignment for the orient and dimension modules, and predicting location + category jointly all lead to worse results than the full model. We also show qualitatively in Fig 4.8 that our strategy help to avoid common failure cases from the previous chapter [125]. Using a separate category module allows our model to generate seats for the living room (left), and introducing a dimension module prevents the use of a too-large cabinet that blocks the office door.

Speed comparisons Table 4.3 shows the time taken for different methods to synthesize a complete scene. It takes on average less than 2 seconds for our model to generate a complete scene on a NVIDIA Geforce GTX 1080Ti GPU, which is two orders of magnitudes faster than the our previous image based method (Deep Synth). While slower than end-to-end methods such as [69], our model can also perform tasks such as scene completion and next object suggestion, both of which can be useful in real time applications.

Perceptual study We also conducted a two-alternative forced choice (2AFC) perceptual study on Amazon Mechanical Turk to evaluate how plausible our generated scenes appear compared those generated by other methods. Participants were shown two top-down rendered scene images side by side and asked to pick which one they found more plausible. Images were rendered using solid colors for each object category, to factor out any effect of material or texture appearance. For each comparison and each room type, we recruited 10 participants, which was sufficient to produce strong 95% confidence intervals. Each participant performed 55 comparisons; 5 of these were "vigilance tests" comparing against a randomly jumbled scene to check that

	Ours vs.		
Room Type	GRAINS [69]	Deep Synth [125]	SUNCG
Bedroom	$\mathbf{82.7 \pm 3.6}$	$\mathbf{56.1 \pm 4.1}$	48.0 ± 4.7
Living	$\mathbf{74.1 \pm 3.8}$	52.7 ± 4.5	45.0 ± 4.5
Bathroom	—	$\mathbf{68.6 \pm 3.9}$	50.0 ± 4.5
Office	—	36.3 ± 4.5	34.8 ± 5.1

Table 4.4: Percentage ($\pm$ 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.

participants were paying attention. We filter out participants who did not pass all vigilance tests.

Table 4.4 shows the results of this study. Our generated scenes are significantly preferred to those generated by GRAINS across all room types (GRAINS does not provide bathroom or office results). Due to format differences, our reconstruction of GRAINs room geometry is imperfect. We manually removed rooms where objects intersect with the walls, but it should be noted that the reconstructed rooms might still differ slightly from the results presented in their work. Compared to Deep Synth, our scenes are preferred for bedrooms and bathrooms, and judged indistinguishable for living rooms. Our generated office scenes are less preferred, however. We hypothesize that this is because the office training data is highly multimodal, containing personal offices, group offices, conference rooms, etc. It appears to us that the rooms generated by Deep Synth are mostly personal offices. We also generate high quality personal offices consistently. However, when the category module tries to sample other types of offices, this intent is not communicated well to other modules, resulting in unorganized results e.g. a small table with ten chairs. Finally, compared to held-out human-created scenes from SUNCG, our results are indistinguishable for bedrooms and bathrooms, nearly indistinguishable for living rooms, and again less preferred for offices.

4.4 Chapter Summary

In this chapter, we presented a new pipeline for indoor scene synthesis using image-based deep convolutional generative models. Our system analyzes top-down view representations of scenes to make decisions about which objects to add to a scene, where to add them, how they should be oriented, and how large they should be. Combined, these decision modules allow for rapid (under 2 seconds) synthesis of a variety of plausible scenes, as well as automatic completion of existing partial scenes. We evaluated our method via statistics of generated scenes, the ability of a classifier to detect synthetic scenes, and the preferences of people in a

forced-choice perceptual study. Our method outperforms prior techniques in all cases.

One of the key limitations of this new approach is that it is still unable to generate room types with multiple strong modes of variation, e.g. single offices vs. conference offices. One possible direction is to explore integrating our image-based models with models of higher-level scene structure, encoded as hierarchies a la GRAINS, or perhaps as graphs or programs. In the next chapter, we present a version of this approach where the high level scene structure is modeled with a relation graph.

Chapter 5

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

In Chapter 3 and 4, we approached scene synthesis in a *space-oriented* fashion: we treat space as a first-class entity, modeling what occupies each point in space. We model space in the form of regular grad i.e. image. In contrast to this paradigm, many other works handle scene synthesis in a *object-oriented* way: they explicitly represent the set of objects in the scene and their properties [34, 93, 152, 69]. 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 chapter, 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 chapter, we believe that PlanIT's "plan-and-instantiate" framework has broader applicability to other layout-generation problems in computer graphics. In chapter 6 and 7, we adopts ideas similar to this to two other domains: 3D floor plans and 3D shapes.

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 the convolutional neural network (CNN)-based scene synthesis modules introduced in Chapter 4 [100]. 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 [70]. 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, we make the following contributions in this chapter:

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`.

5.1 Overview

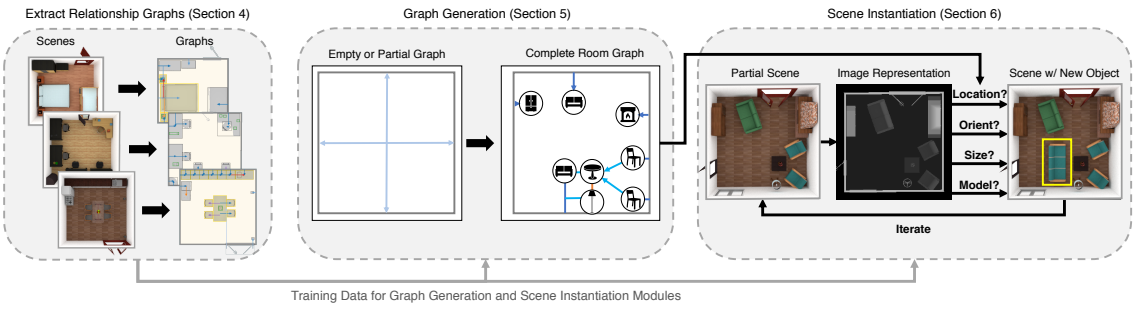


Figure 5.1: Our scene synthesis pipeline. We automatically extract relation graphs from scenes (Section 5.2), which we use to train a deep generative model of such graphs (Section 5.3). In this figure, the graph nodes are labeled with an icon indicating their object category. We then use image-based reasoning to *instantiate* a graph into a concrete scene by iterative insertion of 3D models for each node (Section 5.4).

In this chapter, we aim to build a *indoor scene synthesis* 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 5.1 Left). Section 5.2 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 5.1 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 [70]. Section 5.3 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 5.1 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 the modules from Chapter 4, modifying them to take the graph as input and to attempt to adhere to the structure that it mandates. Section 5.4 describes these modules, and our overall scene instantiation search procedure, in more detail.

5.2 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.

Figure 5.2: Possible types of functional symmetries with which graph nodes can be labeled, along with an example object of that symmetry type.

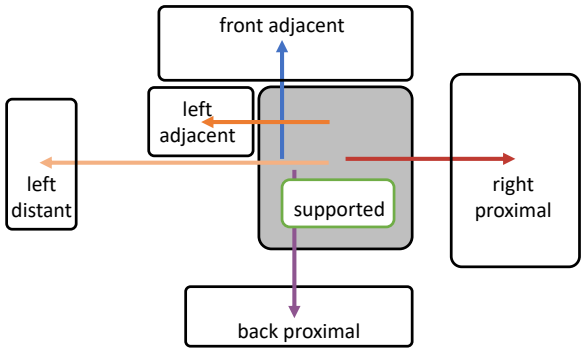


Figure 5.3: Relationships modeled by the edges in our relation graphs. We define *support* edges for statically supported child nodes, and the four spatial edges *front*, *left*, *right*, and *back*, at three distances: *adjacent*, *proximal* and *distant*. The hue of each arrow indicates the relationship direction, and the saturation indicates distance. Supported nodes are outlined in green. We use this same color scheme throughout all figures in this chapter.

5.2.1 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 [114]. 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., in a similar fashion to the previous two chapters. 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 5.2.2). 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.

5.2.2 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 5.2 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 $A \rightarrow B$ 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: $A \rightarrow B$ 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 5.3 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 $\rightarrow$ 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).

5.2.3 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 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 $10\,\mathrm{cm}$. 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 $A \rightarrow B$ 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 5.4 shows an example of each of these types of superstructures,

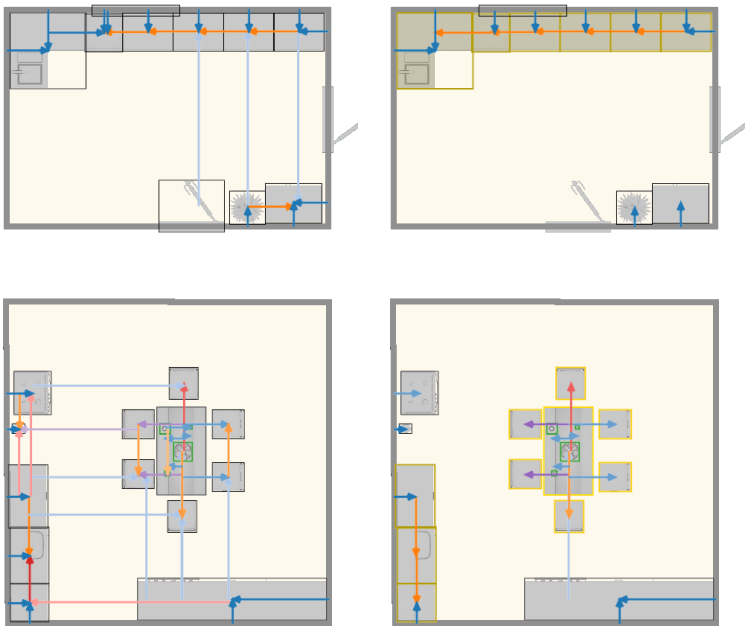


Figure 5.4: Graphs before *(left)* and after *(right)* the detection of superstructures. Hub-and-spoke and chain superstructures are indicated with yellow and brown bounding boxes around member nodes, respectively. Superstructures organize relationships more compactly (e.g. the chains of kitchen cabinets along the walls and the chairs relative to the dining table).

Table 5.1: Average node and edge count statistics for graphs extracted from SUNCG rooms using the automatic procedure from Section 5.2. "Non-wall" edges are those where neither endpoint is a wall.

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

and Appendix A.5 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 [9], 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 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

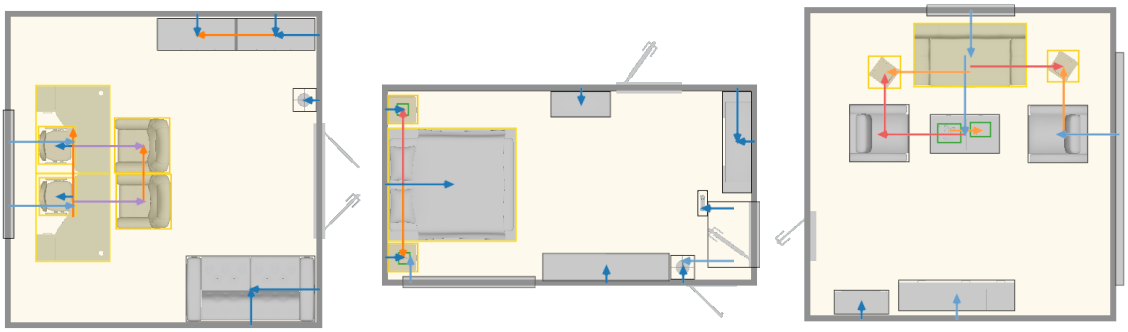


Figure 5.5: Examples of relation graphs extracted from training set scenes.

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.5 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 5.4, 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.5.

Table 5.1 shows some statistics for our extracted graphs. Figure 5.5 shows some example scenes and the graphs extracted from them.

5.3 Graph Generation

Our goal is now to learn a generative model from our extracted graphs. Graph generation is a long-studied problem [30, 102]. 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

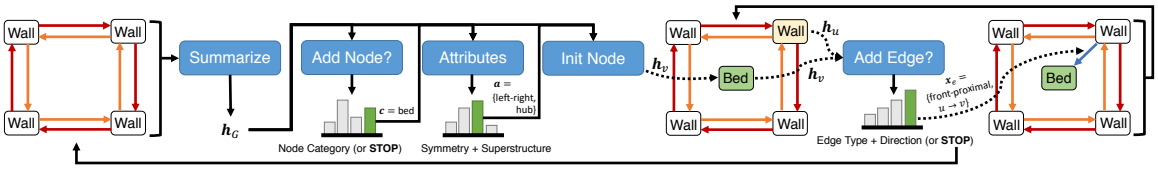


Figure 5.6: Overview of the graph generation pipeline. We start from nodes and edges describing the architecture of the room and iteratively add new nodes and edges with a sequence of decision modules that predicts the category of the new node, the symmetry and superstructure type of the new node, and the edges incident to the newly added nodes.

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.

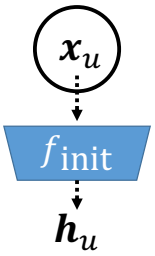
5.3.1 Autoregressive Graph Generation via Message-Passing Graph Convolution

Our approach to graph generation is based on the framework introduced by [70]; 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).

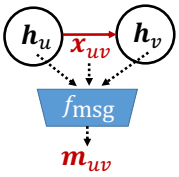
On what basis should the generator should make these decisions? Ideally, it would have some way to "look" at the current state of the graph and make a decision based on what it "sees." For image-based generative models, a convolutional neural network (CNN) provides this capability: a CNN can ingest an 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.

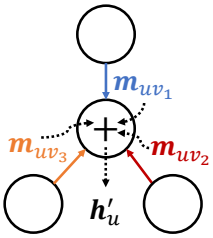


Propagation Similarly to image convolution, graph convolution aggregates information from proximal nodes in the graph. To do so, the following steps are executed:

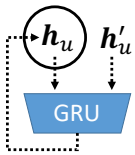
1. Compute edge messages: Information propagation in the graph follows edges, which define proximities in the graph. A message function f_{msg} is used to computed the propagated message $\mathbf{m}_{uv}$ from the edge feature $\mathbf{x}_{uv}$ and the latent node features $\mathbf{h}_u$, $\mathbf{h}_v$. If the edge is directed, the order in which inputs are supplied identifies the direction of the edge.



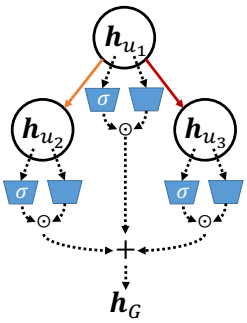
2. Aggregate messages: To actually propagate the computed messages, they are gathered at the nodes incident to their associated edges. Since a node can have varying degree, an operation is needed to map the messages to an aggregated message $\mathbf{h}'_u$ with a fixed dimension. This is often done by 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.



3. 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.

5.3.2 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 [70], 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 5.2 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 5.6 shows our pipeline. Starting with architectural nodes and edges, we use several decision modules to iteratively build the graph:

1. 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 [100]. 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.

2. 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.

3. 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

Table 5.2: Information our model consumes and/or predicts for different types of nodes and edges in the graph.

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

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 [70], 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 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 [70], 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 5.7 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

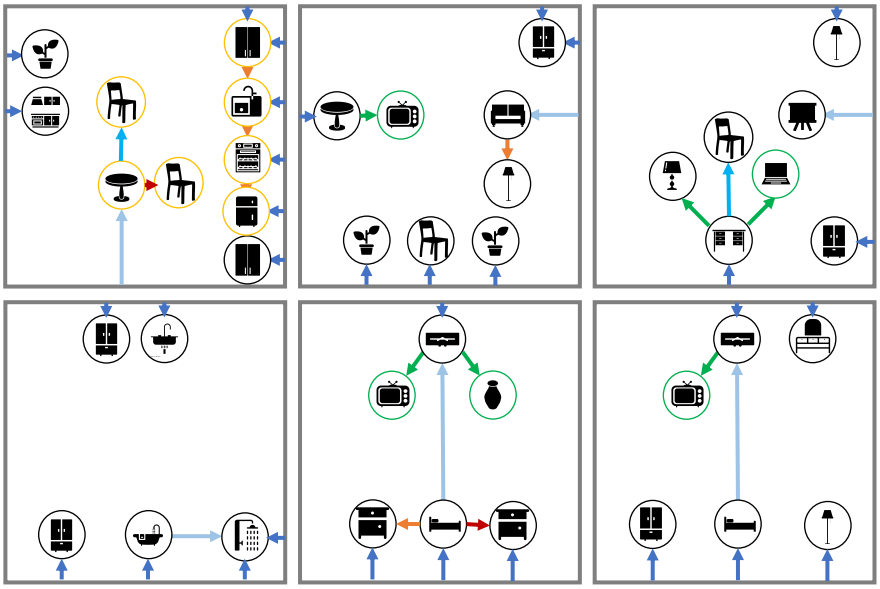


Figure 5.7: Scene relationship graphs generated by our model.

increase global coherence [55]. One could also use reinforcement learning to force the model's output to be spatially realizable [147].

5.4 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 5.2, 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 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:

$$p(\mathcal{S}|\mathcal{G}(V,E)) = p(\mathcal{S}) \cdot \prod_{\mathbf{v} \in V} \mathbb{1}(\mathbf{v} \in \mathcal{S}) \cdot \prod_{(\mathbf{u},\mathbf{v},r) \in E} r(\mathcal{S}, \mathbf{u}, \mathbf{v}) \tag{5.1}$$

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 5.1. This makes our instantiation algorithm a *neurally-guided search* procedure [99, 121].

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.

5.4.1 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 [103]. 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 $\mathcal{C}_{\rightarrow}(\mathbf{v})$ 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 $\mathcal{C}_{\rightarrow}$ score, we break ties based on which node would be *most constraining* if it were to be instantiated next. We define a second score $\mathcal{C}_{\leftarrow}(\mathbf{v})$ as:

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

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.

Similar to the previous chapter, we also order objects by a combination of their size and frequency in the dataset. This 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.

5.4.2 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 5.1. In other words, we need to design an importance sampler for p.

Following the strategy proposed in the previous chapter, 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 use the location prediction module from the previous chapter, using 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 5.1, as many proposed locations will violate the constraints (Table 5.3).

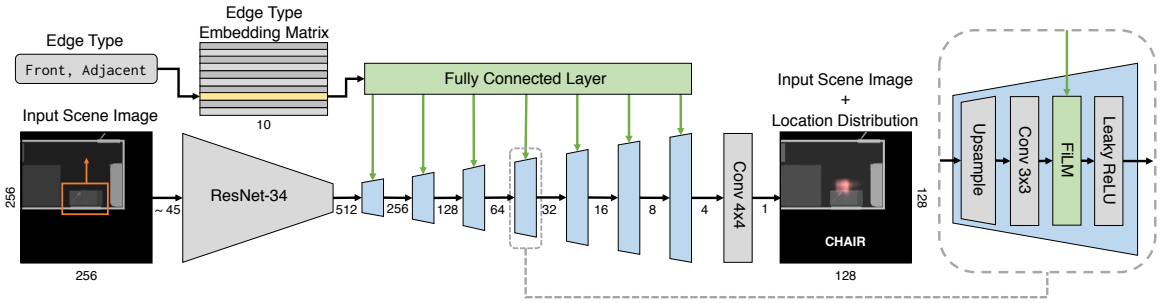


Figure 5.8: Architecture of our graph-conditional location prediction network. We use a fully-convolution network (FCN) architecture to predict a 2D distribution of possible object locations, in the relative coordinate frame of an anchor object (orange region in the input image). The network's output is conditioned on a type of relationship edge via featurewise linear modulation (FiLM) [89]. The network simultaneously predicts distributions for all categories; here we visualize the slice for "chair."

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 to be aware of the relationship graph. Figure 5.8 shows a schematic of this architecture. Rather than predicting a location distribution for a target object in the

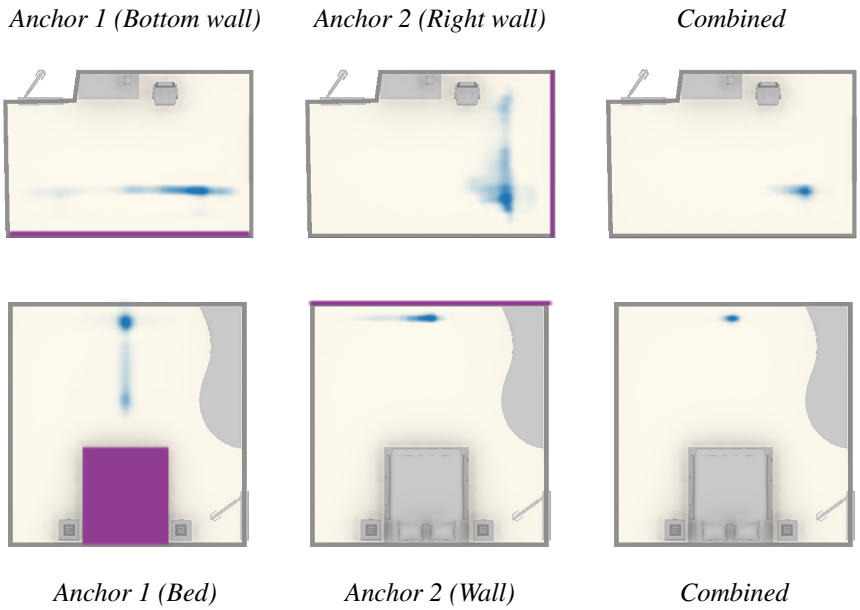


Figure 5.9: Combining multiple anchor-relative location distributions into one global distribution. The anchor object is highlighted in purple. Note how the product of the two anchor-relative distributions leaves an unambiguous signal that the bed should be in the corner *(top row)* and that the TV stand should be directly across from the bed *(bottom row)*.

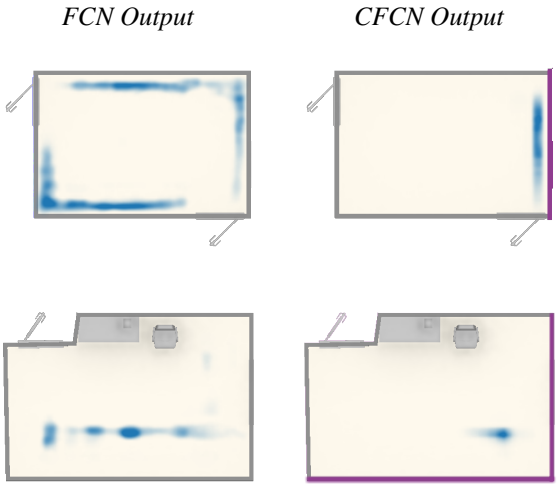


Figure 5.10: Output of the previous unconditional fully convolution network (FCN) compared with our edge-conditional CFCN. The FCN predicts mostly plausible locations, but is oblivious to the structure of the graph and thus likely to suggest constraint-violating locations. Once again, anchor objects (e.g. edge start nodes) ar highlighted in purple in the righthand column.

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 [89]. 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 5.9). This local location prediction strategy can be seen as a form of visual attention [138], 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 5.3, Figure 5.10).

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 the previous chapter. 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 5.4.1 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 A.6 provides more details

Table 5.3: 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)*.

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

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 not (and in general cannot) guarantee that the graphs it outputs are physically realizable. Appendix A.6 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 5.3 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.

5.5 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 1.4 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 5.4, a relationship graph usually does not uniquely specify a scene. To illustrate this visually, Figure 5.11 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, using the same method as that of Figure 3.12. 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 [100]. 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 5.12 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 [69] or must invoke a more expensive optimization process to do so [152].

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 5.4 shows the results of this experiment. PlanIT consistently outperforms GRAINS [69], the previous state-of-the-art object-centric scene synthesis system. When compared to the image-based method from the previous chapter, 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

Figure 5.11: A graph does not uniquely describe a scene; here we show multiple instantiations of the same graph.

Figure 5.12: 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.

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

Table 5.4: Percentage ($\pm$ 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.

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

Table 5.5: Real vs. synthetic classification accuracy for scenes generated by different methods. Lower (closer to 50%) is better. Adapted from [100], Table 2.

Method	**Accuracy**
Deep Synth [125]	84.69
Fast Synth [100]	58.75
Ours	63.13

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 that in Chapter 4 [100]: we use a Resnet34 that takes as input the same top-down scene representation used by the location suggestion FCN from Section 5.4.2, 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 5.5 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 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.

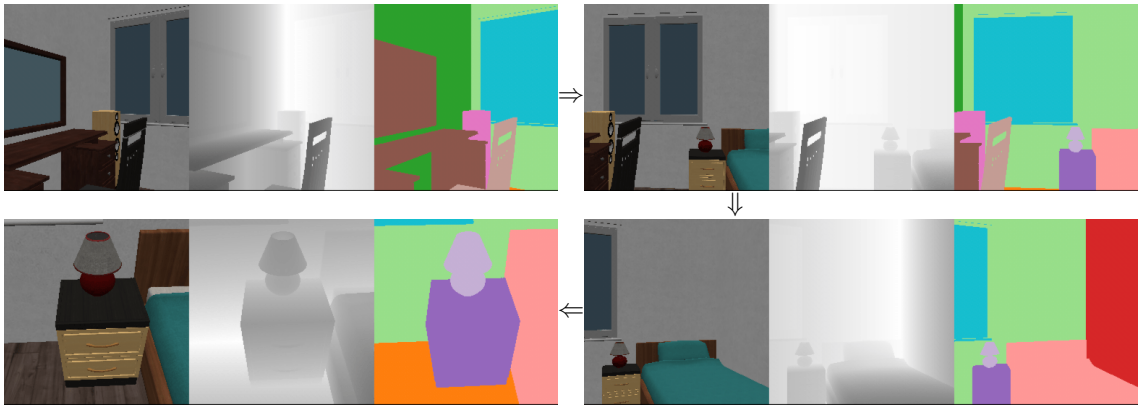


Figure 5.13: Our approach can be used to generate specific, task-relevant scenes for use in 3D simulation. In this example, we generate a bedroom with a nightstand and lamp. Then we use the MINOS [104] simulator to extract frames for color, depth and semantic segmentation during a navigation trajectory where the goal is to locate the lamp on the nightstand (sequence is clockwise from top left to bottom left).

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 $\approx 1.5s$ 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 5.13 shows an example.

5.6 Chapter Summary

In this chapter, 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.

Figure 5.14: Typical failure cases for our model. From left to right: TV stand blocks access to part of the room; this particular desk cannot be accessed when used in a corner; loudspeakers placed behind TV stand instead of beside it; failure to precisely arrange dining chairs around the table.

Our method is not without its limitations. Figure 5.14 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 (right). 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 [51]. 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 [13, 15], including recent work that uses a graph-based intermediate representation [76]. 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.

Chapter 6

Roominoes: Generating Novel 3D Floor Plans From Existing 3D Rooms

In the previous chapters, we introduced a collection of methods that perform *indoor scene synthesis*. Our methods, like many other works [69, 154], learn furniture layouts from synthetic scene datasets [114, 37, 65, 141] and generate new synthetic 3D data. However, regardless of how these synthetic scenes are created, there exists a "reality gap" between synthetic scenes and real ones, both in terms of content (e.g. synthetic scenes are not messy enough) and style (e.g. synthetic scenes are not photorealistic enough) [151].

When tasks demand such realistic scene data, one popular option is going for one of dataset of scanned 3D environments. Unfortunately, the scale and diversity of existing datasets can be a bottleneck: they are not very large nor configurable, and it is time-consuming and expensive to enlarge them. For example, existing datasets provide mostly 'detached' rooms (ScanNet [25]), or are limited to a small number of high-quality multi-room residences (Gibson [136], Matterport3D [12] and Replica [115]).

In this chapter, we propose a new paradigm for creating house-level 3D environments by piecing together existing 3D rooms, a task that we call *Roominoes*. This mix-and-match strategy is similar to room-level 3D scene synthesis that selects from a database of 3D objects and arranges them to create a furnished room. Instead of arranging 3D objects, the Roominoes task requires arranging entire rooms. In other words, the rooms are building blocks to produce a coherent floor plan through a 'retrieve and edit' approach. This approach allows for scalability through combination of existing high-quality 3D rooms, as well as control of the level of complexity (from simple environments with few rooms, to complex environments with many

rooms).

To analyze the challenges presented by the Roominoes task, we decompose it into three sub-tasks: (1) generating a 2D floor plan that is compatible with available 3D rooms, (2) retrieving a set of 3D rooms that can fit into the 2D floor plan, and (3) deforming individual 3D rooms so they fit the 2D floor plan. A Roominoes method must address these three sub-tasks. There are a spectrum of strategies for performing them, each with different tradeoffs. One could use existing 2D floor plans to provide plausible layouts; however, fitting available 3D rooms into those layouts may induce undesirably large deformations to the 3D rooms and the objects they contain. Alternatively, one could directly piece together existing 3D rooms without starting with a target layout; this will result in minimal distortion to the 3D rooms but may produce a less-realistic layout. In this chapter, we implement and evaluate these two strategies. To measure the quality of the generated 3D environments, we introduce a set of evaluation metrics for measuring both the 2D layout quality and 3D mesh quality (e.g. amount of deformation, navigability). We also discuss the potential usefulness of these generated environments for downstream tasks, and we demonstrate the applicability of generated 3D houses on a visual navigation task.

We find that the Roominoes task is challenging, requiring non-trivial geometry processing to properly deform room architectures while keeping contained objects relatively undeformed. We believe that compositional generation of realistic multi-room 3D houses will become increasingly useful.

6.1 Overview

Generating 3D floor plans using existing 3D rooms is a more constrained problem than typical 2D floor plan generation: one cannot generate rooms of arbitrary shapes, but instead is limited to the shapes available in an existing 3D room database. It would be too restrictive, however, to allow no changes to the 3D room geometry at all, as it is unlikely that an existing set of 3D rooms can be recombined perfectly into a novel floor plan. The likelihood of creating plausible layouts from existing rooms can be greatly increased if we allow deformations to the original rooms. By distorting the outline of the room and moving the positions of connecting doors or spaces (which we call *portals*), we can reuse the available 3D rooms in a more flexible manner and generate higher quality layouts. However, this approach comes at the cost of degrading 3D room quality, which decreases as the amount of distortion increases.

There is no definitive answer on how to trade off between 2D layout plausibility and 3D mesh quality; different applications may warrant different tradeoffs. Thus, we explore a spectrum of strategies which span

this tradeoff space. We propose to break down the Roominoes task into three parts: i) generating a 2D floor plan that is compatible with available 3D rooms; ii) retrieving 3D rooms that can fit into the 2D floor plan; and iii) deforming the 3D rooms so they better fit the 2D floor plan. We note that the third step, room deformation, can be solved separately, provided that a correspondence can be found between each retrieved 3D room and its counterpart in the 2D floor plan. Thus, our exploration of possible strategies focuses on approaches for solving the first two steps, taking into account the interaction between them. In the remainder of this section, we first identify strategies for each of these steps and then propose two full pipelines, which we describe further in the following sections.

6.1.1 Generating 2D Floor Plans

3D interiors follow the structure of an underlying 2D floor plan. Taking into consideration that the rooms in a 2D floor plan must be matched with available 3D rooms, we propose the following strategies for obtaining a 2D layout:

Existing Dataset An obvious option for obtaining 2D floor plans is to draw them from an existing floor plan dataset, such as RPLAN [133] (80K annotated floor plans), or LiFULL [1] (millions of floor plan images). These datasets are far larger than their 3D scene counterparts: the Matterport3D [12] dataset, for example, contains only 184 floors. Using an existing floor plan guarantees the quality of the layout, but has other disadvantages. The first issue is lack of control: if one wants to replace a bedroom with a living room, or to make a balcony larger, it is unlikely that another floor plan in the dataset satisfies such demands. The second issue concerns the availability of 3D rooms which match the shape of the rooms in the 2D layouts. In general, 3D rooms with exact shape matches are not available. To fit the 3D rooms into the layout, large deformation is often necessary, which can impact the resulting 3D mesh quality.

Generative Model Instead of using floor plans drawn directly from a dataset, one could also use one of the many available data-driven generative models of 2D floor plans to generate novel 2D layouts [133, 50, 84]. This approach will improve the control over the resulting layout, since these methods usually accept some user input specification (e.g. a room relationship graph) and are capable of generating multiple possible layouts given the same input specification. However, this approach does not address the issue with 3D deformation artifacts.

Table 6.1: Comparing different strategies for generating 2D layouts.

Strategy	Layout Quality	Control	Deformation
Dataset	High	Low	Large
Generative Model	High	Medium	Large
Naive Portal Stitching	Low	High	None
Smart Portal Stitching	Medium	High	Medium

Naive Portal Stitching One can avoid these deformation artifacts entirely by ignoring 2D layout altogether: instead of combining 3D rooms according to a layout, we can instead stitch rooms together by connecting pairs of portals. This approach removes the need to deform 3D rooms, but may produce an incoherent layout and is incompatible with downstream tasks that involve reasoning about the global scene layout.

Smart Portal Stitching Finally, one can attempt to find a "middle ground" trade-off between 2D layout plausibility and amount of 3D deformation by modifying the portal stitching method above: identifying 3D rooms that can likely fit together into a plausible layout without causing too much deformation.

Table 6.1 summarizes the strengths and weaknesses of these different strategies for obtaining a 2D layout.

6.1.2 Retrieving Compatible 3D Rooms

Depending on the strategy used to generate the 2D layout, the task of retrieving 3D rooms can be quite different. If the entire 2D layout is known in advance, then the task becomes finding the 3D rooms that best match the rooms in the layout (in terms of shape and/or portal placements). On the other hand, if the layout is determined on-the-fly (as in the two "Portal Stitching" approaches above), then the task becomes more ill-posed. As there is no 'ground truth' target room shape to match, one must instead retrieve rooms that are both geometrically compatible with the rooms already in the layout *and* and lead to a globally plausible layout.

6.1.3 Deforming 3D Rooms

The final task involves taking the retrieved 3D rooms and deforming them according to the 2D floor plan. The problem can be solved in two steps: first computing a correspondence between the original room and the target room, and then applying a standard mesh deformation algorithm according to the defined correspondence. When the available 3D rooms come equipped with semantic object annotations, one can also leverage this information to avoid introducing visually objectionable artifacts such as non-rigid bending of

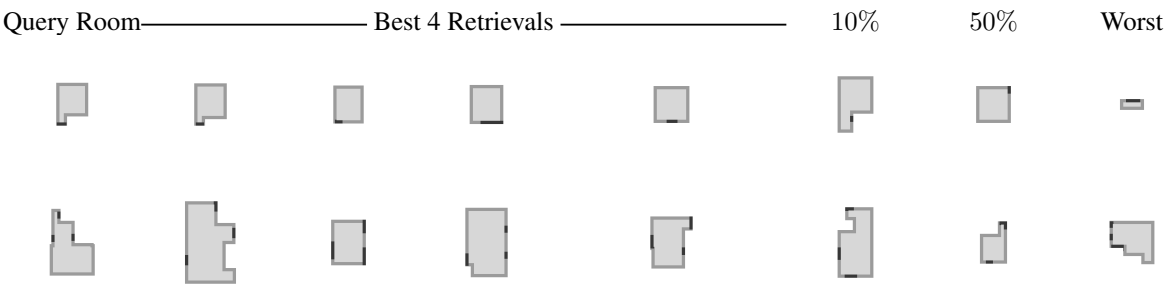


Figure 6.1: Given a 2D room in a floor plan, we find similar 3D rooms that can be deformed into the 2D room with minimal changes with regard to room size, shape, and portal locations. In the top row, the top retrievals are compatible with respect to all criteria, whereas subsequent retrievals start to different in terms of shape, portal location, and finally size. The bottom row shows that the difficulty of finding a compatible 3D room increases substantially when the room shape becomes more non-rectangular, and when the number of portals is greater than 1.

semantically-meaningful rigid objects.

6.1.4 Task Paradigms

While the final task of deforming 3D rooms can be performed independently of the methods used for the first two tasks, these first two tasks are more correlated. We identify two major paradigms for solving them together:

- **2D before 3D:** Generate the 2D floor plan first and use its room shapes as ground truth shape targets for 3D room retrieval (i.e. the "Dataset" strategy in Table 6.1).
- **2D by 3D:** Build up a layout iteratively by retrieving and connecting 3D rooms (i.e. "Guided Portal Stitching" in Table 6.1).

In this chapter, we propose one representative algorithm for each of the two paradigms. In Section 6.2, we describe a method that takes an existing 2D floor plan in the dataset, and retrieves 3D rooms similar according to the floor plan. In Section 6.3, we implement a method that uses deep metric learning to iteratively retrieve compatible 3D rooms and then apply a mixed integer quadratic programming (MIQP) optimization to assemble them into the final layout. Finally, in Section 6.4, we describes a strategy for deforming the 3D rooms, by first optimizing for a set of dense correspondences between the room outlines, and them applying cage deformation to deform the 3D rooms to the target outline.

6.2 Generating 2D Layout Before Retrieving 3D Rooms

In this section, we describe an algorithm that utilizes a large collection of available 2D floor plans to guide the retrieval of 3D rooms. Since the 2D layout is given by a floor plan from the dataset, the focus of this algorithm is on minimizing deformation of the retrieved 3D rooms needed to fit into the 2D floor plan. Since it is computationally expensive to perform deformation for all available 3D rooms, we instead develop approximate metrics to assess how much deformation would be required.

6.2.1 Data Preparation

As our source of 2D floor plans, we use the RPLAN dataset [133], a collection of $80,000$ floor plan images annotated at pixel level. As in prior work, we apply additional filtering to the data: we make all walls the same thickness, retain only the largest portal between each pair of connecting rooms, and remove floor plans containing multi-purpose rooms.

As our source of 3D room data, we use the Matterport3D dataset [12], a collection of 184 floor plans containing 2056 rooms. For each 3D room, we build a 2D representation equivalent to those in the 2D floor plan dataset by extracting its outline (given by human annotations) and using raycasting to identify open regions on its walls, which we consider as portals. Since all of our 2D rooms are rectilinear, we also convert the non-rectilinear architecture of 3D rooms by computing their rectilinear bounding cages (i.e. the union of axis-aligned bounding boxes of all wall segments). We discard rooms that are not closed as well as rooms where a portal falls on a non-rectilinear part of the wall. We also discard room types not present in RPLAN. This leaves us with a final retrieval database of 1036 rooms. Finally, we apply a four way rotational augmentation of this dataset.

6.2.2 Retrieving Compatible 3D Rooms

Given a room from the 2D floor plan, we first filter out a subset of the 3D rooms that are compatible: those that have the same type label and contain same number of portals. For each 3D room in the filtered subset, we compute the following matching score between the source 3D room R_S and the target 2D room R_T, each containing k portals $P_S^0 \dots P_S^{k-1}$ and $P_T^0 \dots P_T^{k-1}$:

$$\lambda_a c_{\text{area}} + \lambda_o c_{\text{outline}} + \lambda_p c_{\text{portal}}$$

where

$$c_{\text{area}} = \frac{\max(A(R_S), A(R_T))}{\min(A(R_S), A(R_T))} - 1$$

penalizes large difference in room areas $A(R_S), A(R_T)$, which will lead to large uniform scaling in the deformation process.

$$c_{\text{outline}} = d_{\text{chamfer}}(O(R_S), O(R_T))$$

computes the two-way chamfer distance between a 250 point sample of the room outlines $O(R_S), O(R_T)$, with the rooms normalized to have unit area and centered. This penalizes room outline differences which lead to nonrigid deformations.

$$c_{\text{portal}} = \min_{j=0}^{k-1} \sum_{i=0}^{k-1} c_{\text{match}}(P_S^i, P_T^{(i+j) \bmod k})$$

tries all possible k ways of pairing the k portals. There are only k ways because the order of the portals along the outline cannot be changed, and pairing any two portals uniquely determines the pairing of the rest. The portal matching cost c_{match} is:

$$\begin{aligned} c_{\text{match}}(P_1, P_2) = (M(P_1) - M(P_2))^2 + \\ \lambda_l(|P_1| - |P_2|)^2 + \\ \lambda_d \mathbb{1}_{\{d(P_1) \neq d(P_2)\}} \end{aligned}$$

where $M(P)$ denotes the position of the midpoint of the portal in the 1D parametrized, unit length, outline of the room, $|P|$ denotes the length of the portal in the same 1D parametrization, and $\mathbb{1}$ is the indicator function that takes the value of 1 if the front-facing directions of the portals $d(P_1)$, $d(P_2)$ are different. This penalizes matching portals that might lead to large deformation: we don't want to slide the portal along the wall by too much, scale it too much, or put it in the wrong orientation. We use $\lambda_a = 1, \lambda_o = 0.5, \lambda_p = 10, \lambda_l = 0.5, \lambda_d = 0.05$ for all experiments.

Figure 6.1 shows examples of rooms retrieved using this score. The score allows retrieval of rooms that are similar regarding size, shape and portal locations. However, due to the limited availability of 3D rooms, it becomes increasingly unlikely to get an exact match as the number of portals and the complexity of the room shape increase.

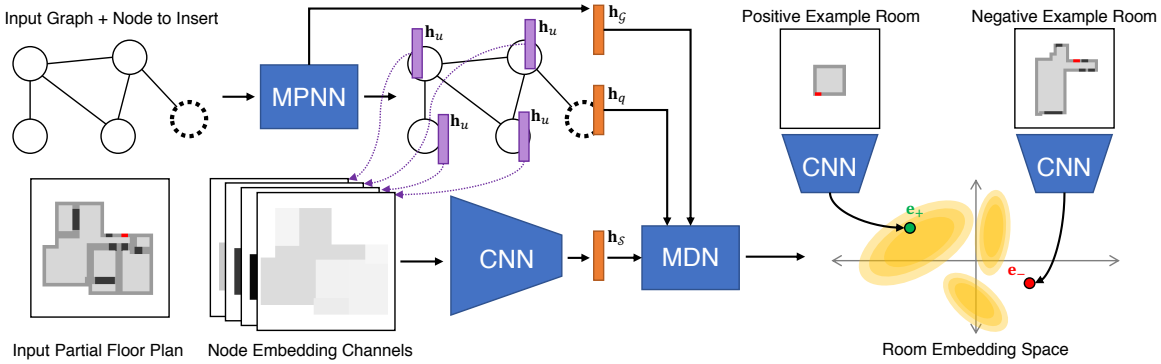


Figure 6.2: The architecture of our neural network-based room retrieval module. The *embedding network* maps top down views of rooms into an embedding vector space (Right). Then, given an input floor plan relationship graph (with the node corresponding to the room to be inserted marked) and a top-down view of the current partial floor plan, the *retrieval network* predicts a Gaussian mixture probability distribution over this embedding space. The two networks are trained jointly such that positive example rooms (obtained by removing random rooms from ground-truth floor plans) have higher probability than negative example rooms (random distractors).

6.3 Generating 2D Layout By Retrieving 3D Rooms

While guiding 3D retrieval with existing 2D floor plans ensures the quality of the resulting scene's 2D layouts, it can lead to large deformations of 3D rooms. To reduce the amount of deformation, it is possible to instead build the layout 'on the fly': retrieving 3D rooms one by one and iteratively assembling them into a plausible 2D layout, changing the rooms slightly as needed. In this section, we describe one pipeline that follows this paradigm. We extract a graph abstraction of an existing 2D floor plan, where each node encodes the type of the room and and each unlabeled edge represents a portal connection. Conditioned on the graph, we iteratively retrieve compatible 3D rooms and use an optimization procedure to assemble them into a partial floor plan, repeat these steps until the layout is complete.

6.3.1 Room Retrieval

Each iteration of the algorithm must retrieve a 3D room corresponding to a node in the input layout graph. The retrieved room needs to be compatible not only with the shapes of the other rooms retrieved so far, but also with the rooms retrieved later on, in order to form a coherent floor plan. Instead of trying to manually select features that lead to such compatibility, we formulate the retrieval problem as a deep metric learning problem: we learn an embedding space of possible room shapes, where rooms which are compatible with the same layouts should be grouped together. This is similar in spirit to the approach taken by Sung et

al. for learning to assemble part-based shapes [117]. Our neural architecture has two sub-networks: an *embedding network* and a *retrieval network*. The embedding network maps individual 3D rooms into a high-dimensional vector space; the retrieval network maps an input partial floor plan, as well as a relation graph of the entire floor plan, into a probability distribution over this space. The two networks are trained jointly using a contrastive learning framework, i.e. rooms that are known to fit well with a layout in the training set should map to higher-probability points than other rooms. Figure 6.2 shows the architecture of these networks and their interaction; the rest of this section describes them in more detail.

Embedding Network The embedding network maps a 2D room r into an embedding vector $\mathbf{e}_r$. For this, we pass an image-based representation of the room through a convolutional neural network (CNN). The image representation uses separate channels to represent the rooms, walls, portals, as well as the specific portal to which we want to connect the next room.

Retrieval Network The retrieval network takes the current partial floor plan $\mathcal{S}$, the target floor plan relation graph $\mathcal{G}$, and the node q corresponding to the room to be inserted, and predicts a conditional probability distribution $P(\mathbf{e}_r \mid q, \mathcal{G}, \mathcal{S})$ over room embeddings $\mathbf{e}_r$. Similarly to the work of Sung et al. [117], we model this distribution as a mixture of Gaussians over a room shape embedding space. We use a mixture density network [8] to predict this distribution.

Since the features of $\mathcal{G}$ and $\mathcal{S}$ are highly correlated, we use a single neural network to learn from both simultaneously. We first use a message passing neural network [41] to extract information from the relationship graph $\mathcal{G}$. After T rounds of message propagation, we predict a per-node descriptor $\mathbf{h}_u$, as well as a descriptor of the entire graph $\mathbf{h}_\mathcal{G}$. To extract information from the partial floor plan, we use a CNN conditioned on an image-based representation of the floor plan to predict an image feature $\mathbf{h}_\mathcal{S}$. The image-based representation is the same as that used by the embedding network, with an additional set of $|\mathbf{h}_u|$ channels. These channels encode the corresponding node embedding $\mathbf{h}_u$ for each room in the layout. Finally, we concatenate the image feature $\mathbf{h}_\mathcal{S}$, the graph descriptor $\mathbf{h}_\mathcal{G}$, as well as the descriptor for the room to be retrieved $\mathbf{h}_q$, and feed it to a mixture density network, which predicts the parameters of the Gaussian mixture, where the k-th Gaussian is parameterized by a weight ϕ_k, a mean μ_k and a standard deviation σ_k.

Joint Training We train the embedding and retrieval networks jointly using a triplet loss. Given a room embedding $\mathbf{e}_r$, we define the log likelihood that the embedding is sampled from the mixture distribution

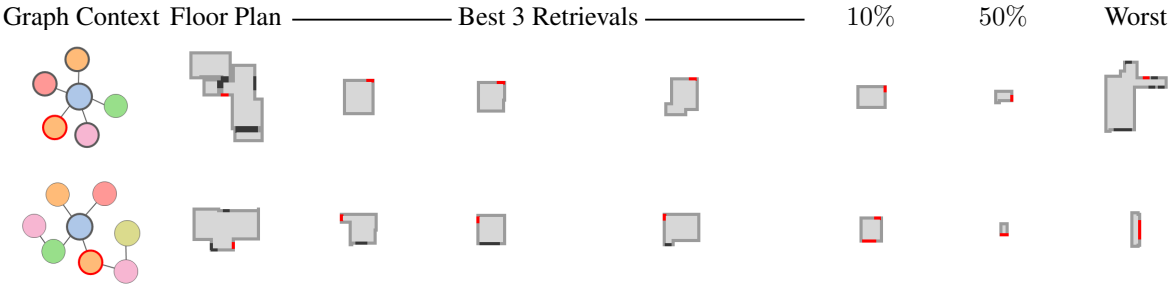


Figure 6.3: Our learned retrieval network retrieves rooms to add to an in-progress layout. The three highest-probability retrievals are good matches and permit the rest of the graph to be completed later. In the top row, the top 3 retrievals are all medium-sized rooms that can be easily inserted into the available corner given their portal placements. In the bottom row, the query node in the graph specifies that the new room should have two portals, and all the top 3 retrieval results do. The quality of retrieval falls off further down the probability-ordered list of rooms in our validation set, exhibiting errors such as incorrect room types, incorrect shapes, and incorrect numbers of portals.

predicted by the retrieval network:

$$\ell^{q,\mathcal{G},\mathcal{S}}(\mathbf{e}_r) = \log \sum_{k=1}^{N} \phi_k(q,\mathcal{G},\mathcal{S}) \cdot \mathcal{N}(\mathbf{e}_r \mid \mu_k(q,\mathcal{G},\mathcal{S}), \sigma_k(q,\mathcal{G},\mathcal{S})^2)$$

Using this embedding loss, we define a contrastive loss [20] between an embedding of a positive example room $\mathbf{e}_+$ and embedding of a negative example room $\mathbf{e}_-$:

$$\mathcal{L}^{q,\mathcal{G},\mathcal{S}}(\mathbf{e}_+, \mathbf{e}_-) = \max\{m + \ell^{q,\mathcal{G},\mathcal{S}}(\mathbf{e}_-) - \ell^{q,\mathcal{G},\mathcal{S}}(\mathbf{e}_+), 0\}$$

where $m = 10$ is a constant margin. In other words, this loss encourages positive example rooms to have a higher predicted likelihood than negative example rooms. Positive example rooms are created by removing a room from one of the training layouts. Negative examples are created by sampling rooms from other floor plans. For rooms with more than one portal, we randomly select a subset of portals to show in the query portal mask.

We use $T = 5$ rounds of propagation for the graph neural network. We choose $|\mathbf{h}_u| = 64$ as the dimension of the per node embedding and $\mathbf{h}_\mathcal{G} = 128$ as the dimension of the graph embedding. The CNN contains seven layers, without batch normalization, which we find to be detrimental to the training process. We use $|\mathbf{h}_S| = 512$ as the dimension of predicted image feature. For the retrieval network, we predict a mixture of 20 Gaussians over an 128 dimensional embedding space. We use a margin of 10 for the triplet contrastive loss. We train the neural networks using the Adam [61] optimizer, and with a batch size of 16. At

each batch, negative examples are sampled from 16 randomly selected floor plans. We start with uniformly selecting from these examples, gradually favoring harder examples as the training proceeds.

Figure 6.3 shows examples of rooms predicted by the retrieval network. The network learns to retrieve rooms of the correct shape, correct type, and the correct number of portals. Note that the networks are not trained to recognize the possibility of using a room after rotations. To allow for this, we apply a four way rotational augmentation to all the candidate 3D rooms instead.

6.3.2 Placing Retrieved Rooms into the Floor Plan

We use the trained embedding and retrieval networks to iteratively retrieve and insert rooms into the layout. We determine the location of the new room by matching the position of the portal to the connecting rooms. When multiple such portals exist, we randomly select one for initialization.

However, it is unlikely that the retrieved room will fit perfectly into the current partial floor plan. We use beam search to increase the chance that we find better fits. To resolve remaining inconsistencies, we apply an optimization procedure after each step of room retrieval. Inspired by prior work [132, 86], we adopt a mixed integer quadratic programming (MIQP) based procedure for layout optimization. The optimization procedure has two goals. The first is to ensure that the partial floor plan is valid, i.e. each room has a positive area, no overlaps exist between different rooms, and all paired portals align with each other. Figure 6.4 shows examples of this process. The second goal is to avoid obvious semantic issues in the floor plan. We tackle one such issue, interior voids in the floor plan, by encouraging the rooms to be adjacent to each other, though additional heuristics could be employed to improve the quality of the 2D floor plans further. Figure 6.5 shows the effect of encouraging room adjacency. These goals must be achieved while minimizing deformation of the room outlines and the portal locations. In the rest of this section, we describe the precise formulation of the objective function and constraints.

Decomposing Rooms into Rectangles

Rooms in our representation are arbitrary rectilinear polygons. It is intractable to define certain important constraints, such as non-overlap, between such polygons. Thus, our first step is to decompose each room into a set of rectangles. We then use constraints to bind these rectangles together so they behave as a continuous room.

In general, the decomposition of a rectilinear polygon into rectangles is not unique. In our implementation, we use the *maximal* decomposition, which is found by constructing a grid over all vertex coordinates

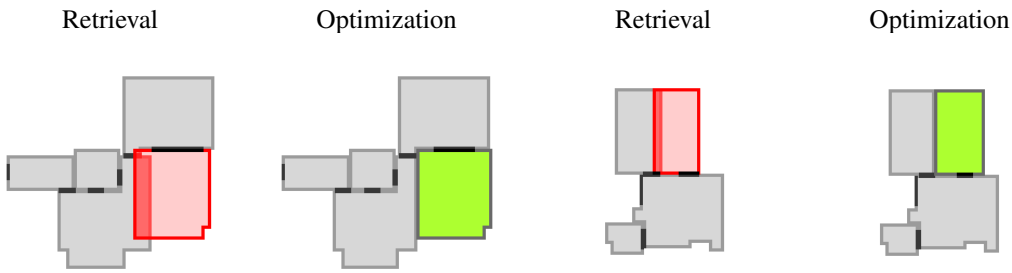


Figure 6.4: Examples of a 2D floor plan before and after our optimization-based snapping. From left to right: fixing a room slightly too large to fit in a small corner; sliding a pair of portals to make the new room match better.

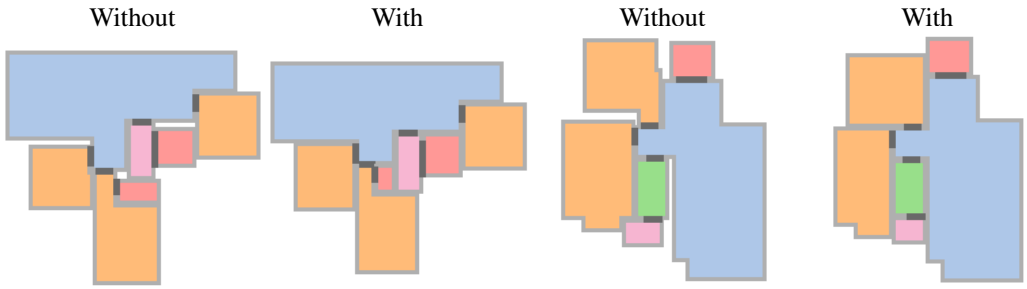


Figure 6.5: Two examples illustrating the effect of rewarding the layout optimizer for maximizing adjacencies between different rooms.

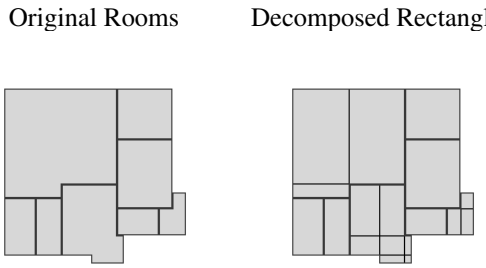


Figure 6.6: Example of a floor plan before and after rectangle decomposition

and then taking the grid cells which fall inside the polygon as the decomposed rectangles (Figure 6.6). The maximal decomposition is quick to compute and also has the property that every pair of adjacent rectangles shares a complete edge. This edge-sharing property makes it easy to impose additional constraints to bind the decomposed rectangles of a given room together.

Optimization Variables

In our solver, we express each rectangle i in terms of four variables: its upper-left vertex position $\langle x_i, y_i \rangle$, its width w_i, and its height h_i. In addition to the room shapes, we must also represent and optimize for the positions and sizes of portals between rooms (e.g. doors), so that the resulting floor plan is navigable. We describe a portal j as a line segment with centroid position $\langle px_j, py_j \rangle$ and radius (i.e. half-length) pr_j. Each portal is permitted to slide along certain walls of the room which contains it, as the next section will describe.

Constraints

Non-negativity All variables must be non-negative so that our solution lies in the positive quadrant and no output rectangles or portals have a (non-physical) negative width or height:

$$x_i, y_i, w_i, h_i, px_j, py_j, pr_j \geq 0 \qquad \forall i, j$$

Minimal Room Size To prevent the optimization from collapsing certain rooms in favor of others, we enforce that each room has a width and height of at least s. To do this, we identify a sequence of indices R^x of rectangles that spans the horizontal extent of the room, as well as a sequence of R^y for rectangles that spans its vertical extent. Then:

$$\sum_{i=0}^{|R^x|} w_{R_i^x} > s, \sum_{i=0}^{|R^y|} h_{R_i^y} > s \qquad \forall i \in r$$

Non-overlap We require the solution to have no overlapping pairs of rectangles i, j. There are four possible relationships to account for: i is either to the top, bottom, left, or right side of j. Let $D \in \{T, B, L, R\}$ represent these relationships respectively. We let the MIQP optimizer select from this set of possible relationships by introducing an auxiliary binary variable $\sigma_{i,j}^D$, where $\sigma_{i,j}^D = 1$ if and only rectangles i and j have

the relationship D. This results in the following set of constraints for each pair of rectangles i, j:

$$
\begin{aligned}
x_i - w_j &\geq x_j - M \cdot (1 - \sigma^R_{i,j}) \\
x_i + w_i &\leq x_j + M \cdot (1 - \sigma^L_{i,j}) \\
y_i - h_j &\geq y_j - M \cdot (1 - \sigma^B_{i,j}) \\
y_i + d_i &\leq y_j + M \cdot (1 - \sigma^T_{i,j}) \\
\sum_{D=1}^{4} \sigma^D_{i,j} &\geq 1
\end{aligned}
$$

where M is a large constant to ensure that rectangles i and j do not overlap in direction D when $\sigma^D_{i,j} = 1$ (we set $M = x_{\text{max}} + y_{\text{max}}$ in our implementation). The last constraint requires that at least one of the four auxiliary variables has a value of 1.

Decomposition constraints For each decomposed room, we introduce the following constraints for all pairs of its rectangles i, j such that i is to the left of j (the rectangles share a vertical edge):

$$x_i + w_i = x_j \qquad y_i = y_j \qquad h_i = h_j$$

and the following constraints for all pairs of rectangles i, j such that i is to the top of j (the rectangles share a horizontal edge):

$$y_i + h_i = y_j \qquad x_i = x_j \qquad w_i = w_j$$

These constraints bind the rectangles together such that they maintain shared edges.

Portal connection If two portals i, j are specified as connected, then their positions and half-lengths must be equivalent:

$$px_i = px_j \qquad py_i = py_j \qquad pr_i = pr_j$$

Portal sliding We require that portals stay on the same wall that they are initially defined to be on, and that their position and length do not extend beyond this wall. If a room decomposes to a single rectangle, then the portal can slide along one edge D of this rectangle (for example, $D = T$ means the portal lies on the top

wall). The sliding constraint thus takes on one of four cases:

$$\text{if D = T} \begin{cases} py_j = y_i \\ px_j \geq x_i + pr_j \\ px_j \leq x_i + w_i - pr_j \end{cases} \qquad \text{if D = B} \begin{cases} py_j = y_i + h_i \\ px_j \geq x_i + pr_j \\ px_j \leq x_i + w_i - pr_j \end{cases}$$

$$\text{if D = L} \begin{cases} px_j = x_i \\ py_j \geq y_i + pr_j \\ py_j \leq y_i + h_i - pr_j \end{cases} \qquad \text{if D = R} \begin{cases} px_j = x_i + w_i \\ py_j \geq y_i + pr_j \\ py_j \leq y_i + h_i - pr_j \end{cases}$$

In general, a room decomposes into multiple rectangles. Here, we must handle the case where a portal lies on a room wall that is shared by more than one decomposed rectangle. This scenario uses the same form of constraint as above, but requires us to know the indices of the rooms between which the portal can slide. For instance, if a portal l slides along the left side of a wall shared by three rectangles i, j, k where i is the top-most rectangle and k is the bottom-most, then the constraints would be:

$$px_l = x_i \qquad py_l \geq y_i + pr_l \qquad py_l \leq y_k + h_k - pr_l$$

Objective

There may be multiple floor plan configurations which satisfy all the constraints defined above. Within this feasible set, there are certain configurations which are preferable. Primarily, we prefer layouts that change room shapes and portal positions/sizes as little as possible, as such changes will introduce distortion when transferred to the 3D mesh.

Secondarily, we prefer layouts which maximize the number of wall-to-wall adjacencies between rooms, as this results in more plausibly compact/space-efficient layouts and also avoids introducing interior voids in the layout. To keep track of adjacenciies, we use a similar formulation to the non-overlap constraint. We add a binary variable $\sigma^A_{i,j}$ for every pair of rectangles (i, j) to indicate whether the two rectangles should be

adjacent, and then we add the following constraints to account for possible adjacency relationships:

$$\begin{cases} x_i \leq x_j + w_j - L \cdot \theta_{i,j} + M \cdot (1 - \sigma^A_{i,j}) \\ x_i + w_i \geq x_j + L \cdot \theta_{i,j} - M \cdot (1 - \sigma^A_{i,j}) \\ y_i \leq y_j + h_j - L \cdot (1 - \theta_{i,j}) + M \cdot (1 - \sigma^A_{i,j}) \\ y_i + h_i \geq y_j + L \cdot (1 - \theta_{i,j}) - M \cdot (1 - \sigma^A_{i,j}) \end{cases}$$

where $\theta_{i,j}$ is a binary variable for whether the rectangles are horizontally or vertically adjacent, and L is the minimum length of the line segment i and j must share to be considered adjacent (we use $L = 6$).

Finally, we minimize the following overall objective function:

where the $\hat{x}$ version of a variable x denotes its initial value. The first term penalizes changes in room rectangle shape; the second penalizes changes in portal radius. The third term rewards pairs of adjacent rectangles from different rooms, but only if the rooms containing those two rectangles i, j have already had all rooms marked adjacent to them in the input graph placed into the layout (this is the role of the indicator function $\mathbb{1}(i, j)$). Finally, the last two terms penalize deviations in all portals' positions along their respective walls. Here, P_{vert} and P_{horz} return the indices of all vertical and horizontal portals, respectively; T_i, B_i, L_i, and R_i give the index of the top, bottom, left, and right adjacent rectangle to portal i's wall, respectively. In our implementation, we use $\lambda_1 = 1, \lambda_2 = 5, \lambda_3 = 100, \lambda_4 = 3$, with all rooms scaled with a ratio of 18 meters to 256 units.

6.4 Deforming Retrieved 3D Rooms

Having generated a 2D layout and retrieved 3D rooms, the final task involves deforming the individual room geometries to fit the layout. This is a nontrivial problem: the 3D room must be warped to fit the new 2D outline, but we must take care to avoid introducing visually-objectionable artifacts, most prominently non-rigid distortion to semantically-meaningful objects within the room.

Deforming a 3D room to fit an optimized floor plan layout amounts to solving an analogy problem: given the source 3D room mesh $\mathcal{M}_S$ and its 2D outline polygon $\mathcal{P}_S$, as well as the target 2D outline polygon $\mathcal{P}_T$ produced by the optimizer, what is the corresponding target 3D room mesh $\mathcal{M}_T$?

In this section, we describe a two-step solution. First, we warp the source outline $\mathcal{P}_S$ onto the target outline $\mathcal{P}_T$ by optimizing for a dense correspondence between them. Then, we use the warped outline as a

control cage for cage-based deformation of the source mesh $\mathcal{M}_S$ to produce the target mesh $\mathcal{M}_T$.

Finding a correspondence between outlines Our first step in deforming the 3D room mesh to fit the new target room outline $\mathcal{P}_T$ is to first deform its 2D outline $\mathcal{P}_S$ to the target outline. As both outlines are closed, one-dimensional, piecewise linear curves, this problem reduces to finding a correspondence between $\mathcal{P}_S$ and $\mathcal{P}_T$. We solve this problem by sampling a finite set of N 1D points $u_S^0 \dots u_S^{N-1}$ along the source outline $\mathcal{P}_S$ (including all of its corner points) and then optimizing for a corresponding set of 1D points $u_T^0 \dots u_T^{N-1}$ on $\mathcal{P}_T$, by minimizing the following objective:

$$\sum_{i=0}^{N-1} \frac{\lambda_e}{N} c_{\text{elasticity}}(i) + \frac{\lambda_n}{N} c_{\text{normal}}(i) - r_{\text{corner}}(i)$$

where

$$c_{\text{elasticity}}(i) = \left((u_T^{(i+1) \bmod N} - u_T^i) - \frac{|P_T|}{N} \right)^2$$

is the elasticity regularization term that encourages the output points u_T^i to be spread out evenly on the target outline,

$$c_{\text{normal}}(i) = \theta_i \left(x_T^{(i+1) \bmod N} - x_T^i \right)^2 + (1 - \theta_i) \left(y_T^{(i+1) \bmod N} - y_T^i \right)^2$$

is the normal matching term that encourages originally vertical segments to stay vertical ($\theta_i = 1$) and originally horizontal segments to stay horizontal ($\theta_i = 0$), where (x_T^i, y_T^i) is the 2D position of u_T^i along $\mathcal{P}_T$, and $r_{\text{corner}}(i) = \sigma_S^i \sigma_T^i$ is the term that encourages source corner points to correspond to target corner points ($\sigma^i \in \{0, 1\}$ is a constant indicating whether the point i is a corner in the source/target).

In addition, we impose the constraint that $u_T^i < u_T^{i+1}$ for all points i, which enforces that the sequence of points remains monotonic. To make sure that the portals fall on the right segment of the outline after deformation, we impose the additional constraint that for each pairs of portals p_S^i and p_T^i, the endpoints of the original portal μ_S^{ia} and μ_S^{ib} falls within the corner points μ_T^{ia} and μ_T^{ib} that contain the target portal. For both the source and target outlines, we define $u = 0$ to occur at the upper-left-most corner of the outline.

Outline-driven deformation Given the warped outline produced by the correspondence step, we drive a cage-based deformation of the room mesh. That is, $\mathcal{P}_S(u_S^0 \dots u_S^{N-1})$ is treated as the initial polygonal control cage for the mesh, and $\mathcal{P}_T(u_T^0 \dots u_T^{N-1})$ provides the new positions of the cage vertices. We then interpolate the positions of all interior mesh vertices using mean value coordinates (MVC) [36]. Finally, we deform each wall where a portal falls on to make sure the portals fall on the right position. Figure 6.7 shows an example

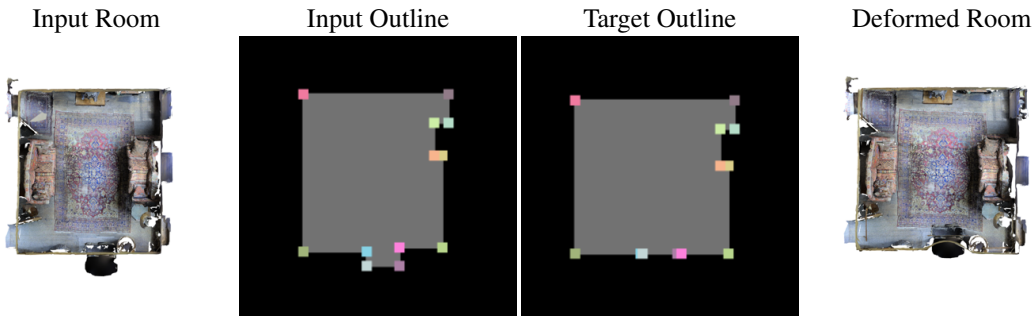


Figure 6.7: Example of a 3D room deformed to fit an optimized 2D outline. From left to right: the initial 3D room mesh; the 2D outline for the input mesh (with a subset of sample points $u_S^0 \dots u_S^{N-1}$ colored); the target 2D outline for the deformation (with corresponding points $u_T^0 \dots u_T^{N-1}$ colored); the final deformed room geometry.

of these procedures applied to a 3D room.

Handling rigid objects The deformation scheme described above non-rigidly distorts the room geometry. Since room retrieval and layout optimization try to retrieve and place rooms that fit into a layout together without changing their shape much, this distortion is typically small and unobjectionable. However, when objects within a room undergo such deformations, it can result in semantically-implausible bending artifacts. To prevent such artifacts, we cut all labeled objects other than those with labels floor, ceiling, wall or curtain out of the mesh, deform the rest of the mesh using the scheme above, and then insert the objects back into the deformed room mesh. To determine object insertion positions, we move their original centroid position according to the MVC deformation. In some cases, this results in placing the object in a position that intersects with other objects or the room geometry. If this occurs, we push the object away from the collision until the intersection is resolved. If doing so results in another intersection before the first is resolved, we uniformly scale the object down until the collision is resolved. Figure 6.8 shows deformation results with and without this special logic for handling rigid objects.

6.5 Evaluation Strategies for 3D Layout Generation

In this section, we propose a set of metrics for evaluating the effectiveness of methods that solve the Roominoes task. First, we identify the properties of the output that are of interest to downstream tasks. We then define metrics that estimate the quality of the results, without having to spend many machine hours to actually test the generated data on downstream tasks. Finally, we demonstrate that 3D scenes generated by Roominoes

Figure 6.8: Examining the effect of treating objects as rigid in our 3D room deformation procedure. From left to right: a view of the initial 3D room mesh; the room deformed without treating objects as rigid; the room deformed while treating objects as rigid. Enforcing object rigidity prevents semantically implausible bending artifacts.

can be useful for downstream tasks.

6.5.1 Evaluating 2D Layout Quality

The quality of a generated floor plan's 2D layout is important for tasks that require reasoning about the global structure of a scene, e.g. indoor navigation. We propose the following metrics to evaluate 2D layout quality:

- **Fréchet Distance (FD)**: a measure of distributional similarity between the generated floor plans and those in a held-out test set [49]. We evaluate this distance in two different feature spaces: a pre-trained Inception image classifier, i.e. the standard Fréchet Inception Distance (**FID**), and the CNN component of the retrieval network described in Section 6.3 (**FD[R]**). The first is more general, whereas the second is domain-specific.

- **Classifier Fool Percentage**: the percentage of held-out test layouts classified as "generated" by a classifier trained to distinguish between generated 2D layouts and layouts from the training dataset. A value of 50% would suggest that the two sets of layouts are indistinguishable from each other.

We compare the following sources of 2D floor plans:

- **Rectangles:** Randomly place N rectangles, where N equals the number of nodes in the graph. This simple baseline establishes a lower bound on performance.

- **Dataset:** Floor plans drawn from a dataset (RPLAN [133]) that are not used for training the following models.

Table 6.2: Comparing different methods for 2D floor plan generation.

Method	FID ⇓	FD[R] ⇓	% fool ⇑
Rectangles	63.49	450.74	1.34
Dataset	6.84	1.29	53.54
Generative Model	9.36	2.72	46.86
Smart Portal Stitching	18.61	86.05	28.72
Smart Portal Stitching (no net)	22.40	90.20	25.00

- **Generative Model:** Generate floor plan given the polygonal outline of the building, using a recent method [133].
- **Smart Portal Stitching**: Build a 2D layout while retrieving 3D rooms using the algorithm described in Section 6.3.
- **Smart Portal Stitching (no net)**: Use the algorithm from Section 6.3, but do not use the learned retrieval network; instead, retrieve random rooms with correct type and portal number.

Table 6.2 shows the results of this experiment. As expected, sourcing 2D layouts from a floor plan dataset or a data-diven 2D floor plan generative model produces the highest quality layouts. The smart portal stitching methods sacrifice layout plausibility, although the version with the learned retrieval network performs better. Figure 6.9 shows examples of dataset layouts vs. those produced by smart portal stitching. The layouts generated by the smart portal stitching method have less consistent outlines and sometimes contain implausible features, such as interior holes. In extreme cases, the layout quality can be too low to be useful, as visualized in Figure 6.10a and 6.10b.

6.5.2 Evaluating 3D Mesh Quality

Large deformations of 3D room meshes will degrade perceived visual quality of the output scenes, which is important to most downstream tasks. We propose the following metrics to evaluate mesh quality in the presence of deformations:

- **Area Change (Area):** Percent change in the area of the room's 2D outline after deformation.
- **Outline Change (Outline):** Average distance between corresponding points on the outlines
- **Portal Change (Portal):** Average distance between the midpoint of the portal after the deformation and the final portal location (obtained through deforming the wall segment)

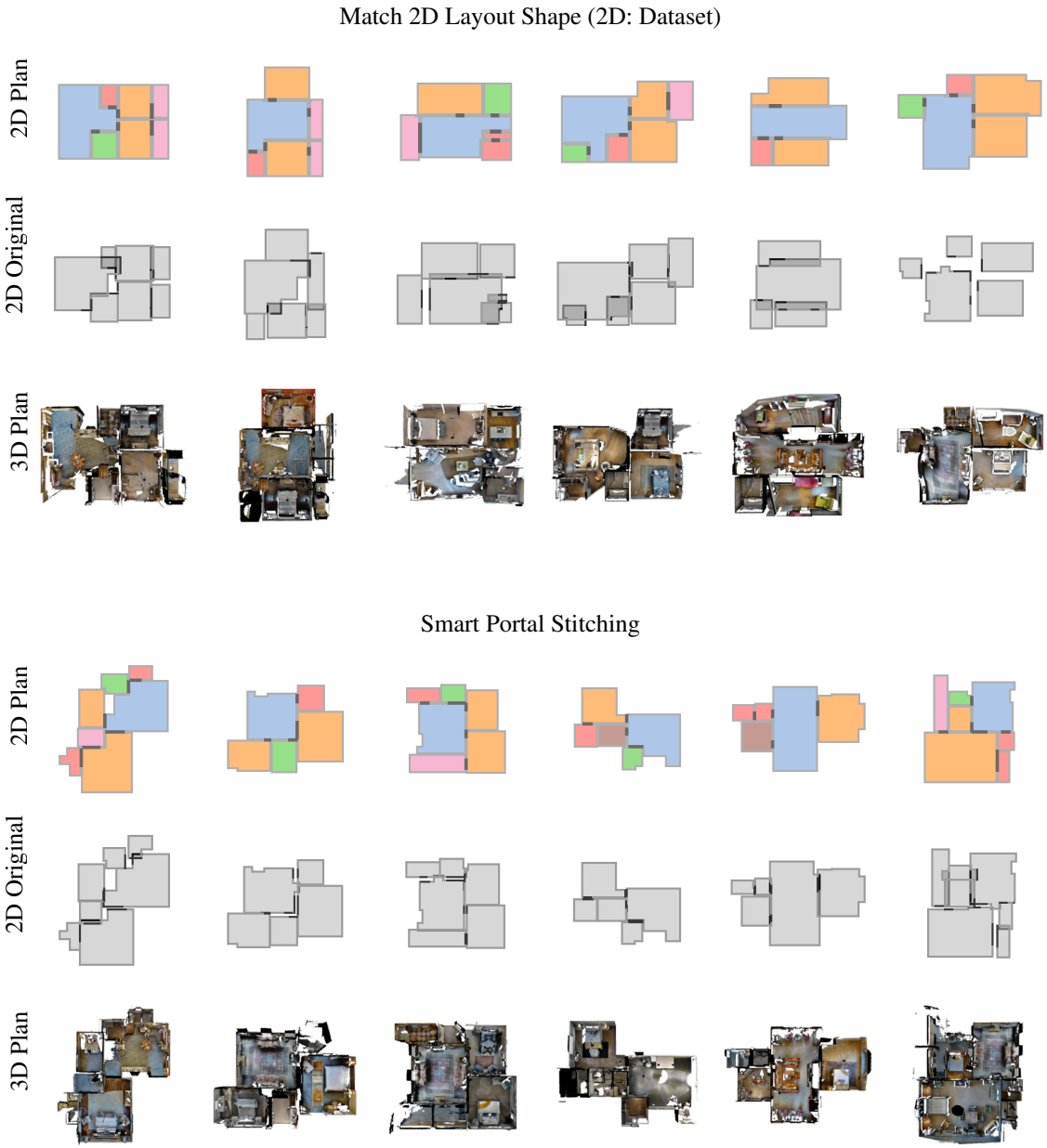


Figure 6.9: Novel 3D house meshes generated by the two representative strategies, visualized along with the 2D floor plan, and an overlay of the original room shapes and portal location. The Match 2D Layout Shape strategy obtains better layouts by directly using existing 2D floor plans. However, the 3D rooms do not match the floor plan well, leading to large deformations and visual artifacts. In contrast, the Smart Portal Stitching strategy requires minimal deformation to the retrieved 3D room outlines and portals, but at the cost of lower layout quality.

Table 6.3: Comparing different methods for retrieving 3D rooms in terms of their effect on final mesh quality. Lower is better.

Method	Area	Outline	Portal	Mesh A	Mesh E
Match 2D Layout Shape	16.31	7.776	2.758	4.47	3.19
Smart Portal Stitching	10.45	0.026	0.008	2.42	1.83
Smart Portal Stitching (no net)	19.32	0.460	0.752	3.47	2.30

Table 6.4: Evaluating how well a pretrained DDPPO [128] agent navigates scenes generated from different sources. Higher is better.

Scene Source	Success Rate	SPL
Match 2D Layout Shape	0.783	0.608
Smart Portal Stitching	0.759	0.618
MP3D Scenes	0.876	0.785

Table 6.5: Navigation agent performance for agents trained on scenes generated using our approach and original Matterport3D dataset scenes. Higher values are better. Evaluation is done on the Gibson [136] dataset, in scenes unseen at training time.

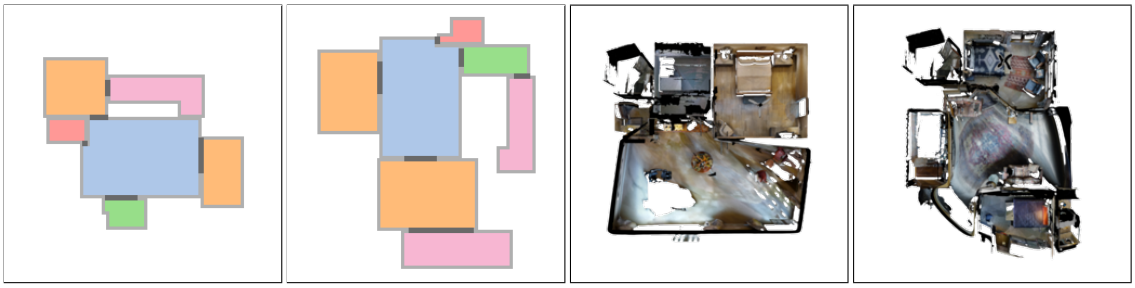
Scene Source	Success Rate	SPL
Smart Portal Stitching	0.753	0.484
MP3D Scenes	0.818	0.628

- **Mesh Statistics**: For each mesh, we discretize the area of each triangle and the length of each edge into 64 bins, and compute the Wasserstein distance of the triangle area distribution (**Mesh A**) and edge length distribution (**Mesh E**).

We use these metrics to compare the following 3D room retrieval methods:

- **Match 2D Layout Shape:** Retrieve rooms whose shape best matches the shape of their corresponding room in a given input 2D floor plan (i.e. the algorithm from Section 6.2).
- **Smart Portal Stitching:** Same as above.
- **Smart Portal Stitching (no net):** Same as above.

Table 6.3 shows the results of this experiment. The smart portal stitching strategy requires significantly fewer changes to the retrieval 3D rooms, with respect to the size, outline shape, portal locations, and mesh properties. Using a neural network further amplifies this effect. On the other hand, due to the limited availability of 3D rooms, the match and deform strategy struggles to find 3D rooms that match exactly, and thus has to modify each room much more. Figure 6.9 illustrates the differences in terms of amount of change



(a) (b) (c) (d)

Figure 6.10: Failure cases of different algorithms. (a, b): the Smart Portal Stitching strategy generates a low quality 2D layout that contains large holes or non-smooth outlines; (c, d): the Match and Deform strategy fails to find reasonably similar living rooms, leading to large deformations in the final mesh.

required and impact on the 3D mesh quality. In extreme cases, the retrieved rooms are so incompatible that leads to unacceptable mesh quality post deformation. Figure 6.10c and 6.10d shows two such examples.

6.5.3 Evaluating on Downstream Tasks

To demonstrate that the 3D house scenes we generate through Roominoes can be useful for downstream tasks, we use them for two small-scale indoor visual navigation experiments. For the first experiment, we use a DDPPO agent [128] trained on the Gibson [136] dataset in the Habitat simulation environment [106]. We task the agent with performing a series of point goal navigation tasks in scenes generated by our method, as well as in original Matterport3D scenes. To evaluate how well the agent navigates these scenes, we use two standard metrics from the visual navigation literature: Success Rate and Success Weighted by Path Length (SPL) [2]. The pre-trained agent can navigate reasonably well in the scenes generated by both methods. Additional qualitative examples of agents navigating in the generated scenes are included in Appendix B.4. For the second experiment, we train two agents with proximal policy optimization [108] with the settings used in [106], one on 104 floor plans generated by the Smart Portal Stitching strategy, and the other on 184 floor plans from Matterport3D. Both agents are trained with 30 million steps. We then evaluate the performance of the agents on the Gibson [136] validation set as used in the Habitat challenge [106]. Table 6.5 summarizes the results. Agents trained on our data achieve comparable success rate to the agent trained on the original Matterport3D data, though with relatively lower SPL. These experiments suggest that data generated by our method is of comparable quality to original Matterport3D data with respect to indoor visual navigation. A full-scale evaluation of the impact of data augmentation with our data is computationally challenging, as it has been shown that navigation agents continue to learn from data after billions of steps [128]. Apendix B.6

reports on an experiment intended to approximate this setting; we leave rigorous, full-scale evaluation to future work. Regardless, it has been shown that additional training data is beneficial to the performance of agents, even if the additional data is of poor quality [128]. Thus, we believe that the scenes we generate, which we have shown to be of reasonable quality, can be used to improve the performance of navigation agents further.

6.5.4 Timing

It takes on average about 30 seconds for the Match 2D Layout Shape strategy to complete a 2D layout. The Smart Portal Stitching strategy, in contrast, requires about 10 minutes per floor plan. Virtually all of the time is spent on 2D layout optimization, where per-room-insertion optimization time varies considerably depending upon layout complexity (from 0.1 to 60 seconds, which is our hard time limit). We note that the time spent on optimization can be reduced from 10 minutes down to 1 minute without beam search. Computing outline correspondence takes about 1 minute per floor plan with 250 point samples. This can be reduced drastically if done with fewer point samples. Finally, deforming the 3D mesh takes 3 minutes per floor plan, primarily due to the large number of vertices we have to handle.

6.6 Chapter Summary

In this chapter, we presented Roominoes, a new task for creating house-level 3D environments by mixing-and-matching existing 3D rooms. We examined the possible solutions to three sub-tasks, and implemented two representative algorithms for solving the task. As discussed in Section 6.5, we found Roominoes to be challenging, primarily due to the need to trade-off between 2D layout quality and 3D mesh quality. There are several potential directions one can take to address such challenges. First, the 2D floor plan quality for the "2D by 3D" class of methods can be improved by designing additional semantic features capturing what constitutes a good floor plan. For example, one could attempt to reduce the number of corners of the generated 2D floor plans to make them more regular, or fine-tune the shape of individual rooms to make the floor plan more similar to real ones. The 3D room deformation procedure can also be improved. Our current method does not fare well in the case of large deformations, particularly when portals need to slide significantly along walls. Future work might instead perform additional mesh surgery e.g. cutting and pasting the portal regions to a different part of the wall, instead of deforming the entire wall, or copying and pasting some existing regions, instead of applying large deformations to those regions. Doing so leads to the need to synthesize

new textures, which is a challenging problem on its own. Currently, there is no guarantee that the layout of the 3D room post-deformation is still realistic. Thus, it would be beneficial to augment the control points with semantic layout rules learned from available 2D layout data. This can be extended further to take into account relations between individual rooms. It is possible to experiment with additional 3D room datasets such as ScanNet [25] or Gibson [136], in order to increase the chance of retrieving rooms that fit the target floor plan well. Finally, a more thorough evaluation in downstream tasks such as visual navigation, as well as other tasks that benefit from 3D scene datasets can reveal what properties of the generated scenes are most beneficial in such tasks.

Beyond addressing these limitations, there are several broader avenues for future work. For instance, one can extend Roominoes to support on-demand data generation, creating scenes tailored to address scenarios where methods for downstream tasks are under-performing. It is also worth doing more investigation on strategies that bridge the gap between different datasets: in this work, we simply combined one dataset of 2D floor plans with another dataset of 3D rooms. While the performance is acceptable, a more careful investigation addressing differences between datasets can potentially improve the generalization performance of the algorithms.

Finally, we would be excited to see Roominoes applied at scale to more of the applications that originally motivated its creation: computer vision, scene understanding, 3D reconstruction, and embodied AI. There are a number of interesting questions for future work. How do we find the right trade-off between 2D and 3D quality depending on the type of downstream tasks? Does training models for these applications on massive sets of Roominoes-generated 3D floor plans improve their generalization performance? And is it possible to reduce the number of training scenes needed if those scenes are tailored to the task(s) the method must address? These are some exciting questions that would be fruitful to explore with Roominoes.

Chapter 7

The Shape Part Slot Machine:Contact-based Reasoning for Generating 3D Shapes from Parts

In the previous chapters, we proposed algorithms for autoregressive of 3D scenes. The 3D shapes that comprise those scenes are also of great interest across multiple fields. In this chapter, we switch the focus to these 3D shapes. There has been a plethora of recent works that focused on unstructured generation of 3D shapes, Recent work in this space has focused on deep generative models of shapes in the form of volumetric occupancy grids, point clouds, or implicit fields. While these methods demonstrate impressive abilities to synthesize the bulk shape of novel objects, the local geometry they produce often exhibits noticeable artifacts: oversmoothing, bumpiness, extraneous holes, etc. At present, none of these generative models has achieved geometric output quality resembling the shapes they are trained on. An alternative approach would be to avoid synthesizing novel geometry altogether and instead learn how to re-combine existing high-quality geometries created by skilled modeling artists. This paradigm is known in the computer graphics literature as *modeling by assembly*, where it once received considerable attention. Since the deep learning revolution, however, the focus of most shape generation research has shifted to novel geometry synthesis. The few post-deep-learning methods for modeling by assembly have shown promise but have not quite lived up to it: handling only coarse-grained assemblies of large parts, as well as placing parts by directly predicting their world-space poses (leading to 'floating part' artifacts).

In this chapter, we present a new generative model for shape synthesis by part assembly which addresses these issues. Our key idea is to use a representation which focuses on the connectivity structure of parts. This choice is inspired by several recent models for novel geometry synthesis which achieve better structural coherence in their outputs by adopting a part-connectivity-based representation [81, 56, 144]. In our model, the first-class entities are the regions where one part connects to another. We call these regions *slots* and our model the *Shape Part Slot Machine*.

In our model, a shape is represented by a graph in which slots are nodes and edges denote connections between them. We define shape synthesis as iteratively constructing such a graph by retrieving parts and connecting their slots together. We propose an autoregressive generative framework for solving this problem, composed of several neural network modules tasked with retrieving compatible parts and determining their slot connections. Throughout the iterative assembly process, the partial shape is represented only by its slot graph: it is not necessary to assemble the retrieved parts together until the process is complete, at which point we use a gradient-descent-based optimization scheme to find poses and scales for the retrieved parts which are consistent with the generated slot graph.

We compare the Shape Part Slot Machine to other modeling-by-assembly and part-connectivity-based generative models. We find that our approach consistently outperforms the alternatives in its ability to generate visually and physically plausible shapes.

In summary, our contributions in this chapter are:

- The *Slot graph* representation for reasoning about part structure of shapes.
- An autoregressive generative model for slot graphs by iterative part retrieval and assembly.
- A demonstration that local part connectivity structure is enough to synthesize globally-plausible shapes: neither full part geometries nor their poses are required.

7.1 Overview

Assembling novel shapes from parts requires solving two sub-problems: finding a set of compatible parts, and computing the proper transforms to assemble the parts. These tasks depend on each other, e.g. replacing a small chair seat with a large one will shift the chair legs further away from the center. Instead of solving these sub-problems separately, we propose a system for solving them jointly. Specifically, our system synthesizes shapes by iteratively constructing a representation of the contacts between parts. The assembly transformations for each part can then be computed directly from this representation.

In our system, a shape is represented as a *slot graph*: each node corresponds to a "slot" (part-to-part contact region) on a part; each edge is either a *part edge* connecting slots of the same part or a *contact edge* connecting slots on two touching parts. Section 7.2 defines this graph structure and describes how we extract them from available data.

This reduces the task of assembling novel shapes to a graph generation problem: retrieving sub-graphs representing parts from different shapes and combining them into new graphs. We solve this problem autoregressively, assembling part sub-graphs one-by-one into a complete slot graph. At each iteration, given a partial slot graph, our system inserts a new part using three neural network modules: the first determines *where* a part be should connect to the current partial graph, the second decides *what* part to connect, and third determines *how* to connect the part. We describe this generation process in Section 5.3.

Finally, given a complete contact graph, the system runs a gradient-based optimization process that assembles parts into shapes by solving for poses and scales of the individual parts such that the contacts implied by the generated slot graph are satisfied. We describe the process in Section 7.4.

7.2 Representing Shapes with Slot Graphs

In this section, we define slot graphs, describe how we extract them from segmented shapes, and how we encode them with neural networks.

7.2.1 Slot-based Graph Representation of Shapes

A good shape representation that models how parts connect allows the generative model to reason independently about part connectivity and part geometry. Given a shape S and its part decomposition $\{P_1 \dots P_N\}$, we call regions where parts connect "slots", and use them as nodes in a graph $\mathcal{G} = (V, E_c, E_p)$, as illustrated in Figure 7.1. Each pair of parts may be connected with multiple slots, and each slot $\mathbf{u}_{ij} \in V$ on part P_i that connects to P_j has a corresponding slot $\mathbf{u}_{ji}$ on part P_j that connects back to P_i. Each node $\mathbf{u}_{ij}$ stores the following properties:

- The axis-aligned bounding box (AABB) of the slot, in a coordinate frame centered on P_i.
- The same AABB, but normalized such that the bounding box of the entire part is a unit cube. This provides a scale-invariant view of how parts connect.

A slot graph $\mathcal{G}$ has two types of edges: *contact edges* $\mathbf{e}^c_{ij} \in E_c$ connect every pair of contacting slots $\mathbf{u}_{ij}, \mathbf{u}_{ji}$ and *part edges* $\mathbf{e}^p_{ijk} \in E_p$ connect every pair of slots $\mathbf{u}_{ij}, \mathbf{u}_{ik}$ in the same part P_i.

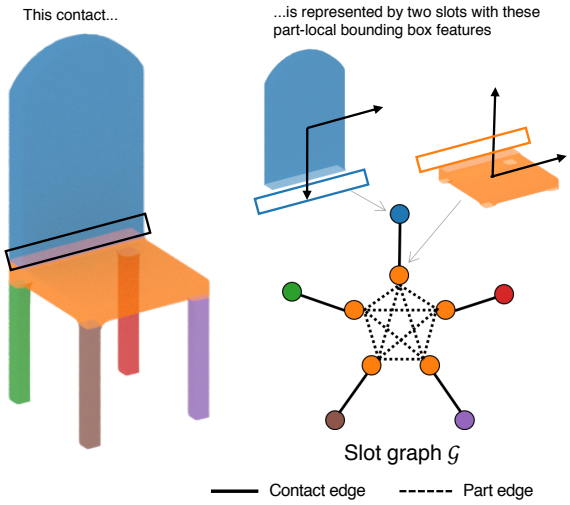


Figure 7.1: A slot graph. Nodes are part-to-part contact regions called *slots* and describe the contact geometry with bounding boxes. *Contact edges* connect two slots on two adjacent parts, while *part edges* connect all slots of the same part.

This representation encodes neither the geometry nor the pose of each part. Omitting this information encourages generalization: the choice of parts will be based only on the compatibility of their attachment regions and connectivity structure; it will not be biased by a part's world-space position in its original shape nor its complete geometry.

This representation also does not encode part symmetries; nevertheless, we demonstrate in Section 3.4 that our model often generates appropriately symmetrical shapes. We can also optionally include logic that enforces symmetries at test time (see Section 5.3).

7.2.2 Extracting Slot Graphs from Data

Given a set of part-segmented shapes, we follow StructureNet [81] to extract part adjacencies and symmetries. We use adjacencies to define slots, and define connecting regions as points within distance τ of the adjacent part. Additionally, we ensure that symmetrical parts have symmetrical slots and prune excess slots where multiple parts overlap at the same region. See Appendix A.7

7.2.3 Encoding Slots Graphs with Neural Networks

We encode a slot graph into a graph feature $h_{\mathcal{G}}$ and per-slot features $h_{\mathbf{u}}$ using messaging passing networks [41].

Initializing Node Embeddings: We initialize slot features $h_{\mathbf{u}}$ using a learned encoding of the slot properties $x_{\mathbf{u}}$ (two six-dimensional AABBs) with a three-layer MLP f_{init}: $h^0_{\mathbf{u}} = f_{\text{init}}(x_{\mathbf{u}})$. As we discuss later, some of our generative model's modules also include an additional one-hot feature which is set to one for particular nodes that are relevant to their task (effectively 'highlighting' them for the network).

Graph Encoder: The node embeddings are then updated with our message passing network using an even number of message passing rounds. In each round, node embeddings are updated by gathering messages from adjacent nodes. We alternate the edge sets E during each round, using only part edges $E = E_p$ for odd rounds ($t = 1, 3, 5 \ldots$) and only contact edges $E = E_c$ for even rounds ($t = 2, 4, 6 \ldots$):

$$h_{\mathbf{t}} = f^t_{\text{update}}\Big(h^{t-1}_{\mathbf{u}}, \sum_{\mathbf{uv} \in E} f^t_{\text{msg}}(h^{t-1}_{\mathbf{u}}, h^{t-1}_{\mathbf{v}}, h^0_{\mathbf{u}}, h^0_{\mathbf{v}})\Big)$$

where f_{msg} is a multi-layer perceptron (MLP) that computes a message for each pair of adjacent nodes, and f_{update} is a MLP that updates the node embedding from the summed messages. We also include skip connections to the initial node embeddings $h^0_{\mathbf{u}}$. All MLPs have separate weights for each round of message passing.

Gathering Information from the Graph: To obtain the final node features $h_{\mathbf{u}}$, we concatenate its initial embedding with as its embeddings after every even round of message passing (i.e. those using contact edges) and feed them into an MLP f_{node}:

$$h_{\mathbf{u}} = f_{\text{node}}(h^0_{\mathbf{u}}, h^2_{\mathbf{u}} \ldots h^T_{\mathbf{u}})$$

To obtain the feature $h_{\mathcal{G}}$ of an entire graph, we first perform a graph readout over the embeddings at round t:

$$h^t_{\mathcal{G}} = \sum_{\mathbf{u} \in V} (f_{\text{project}}(h^t_{\mathbf{u}}) \cdot f_{\text{gate}}(h^t_{\mathbf{u}}))$$

where f_{project} projects node features into the latent space of graph features and f_{gate} assigns a weight for each of the mapped features. We then compute the final graph feature $h_{\mathcal{G}}$ similar to the way we compute the node features:

$$h_{\mathcal{G}} = f_{\text{graph}}(h^0_{\mathcal{G}}, h^2_{\mathcal{G}} \ldots h^T_{\mathcal{G}})$$

In cases where we need a feature for a subset of nodes $V' \subset V$ in the graph, we simply perform the readout over V' instead of V.

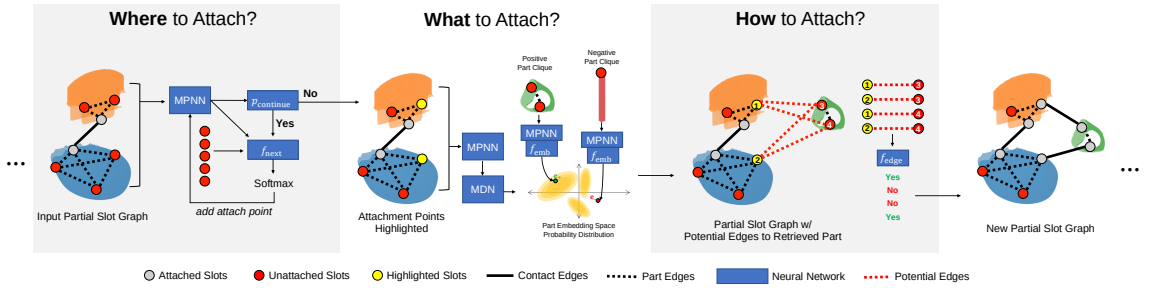


Figure 7.2: Our slot graph generative model uses three neural network modules to build a graph step by step. *Where to Attach?*: Predicts which slots on the current partial shape the next-retrieved part should be attached to. *What to Attach?*: Learns an embedding space for part slot graphs and predicts a probability distribution over this space in which parts which are compatible with the highlighted slots have high probability. *How to attach?*: Determines which slots on the retrieved part should connect to which slots on the current partial shape.

7.3 Generating Slot Graphs

Our system casts shape synthesis by part assembly as a problem of generating novel slot graphs. To do so, we first extract a graph representation of each part: for every shape S in a dataset of shapes, and for every part $P_i \in S$ represented by a slot graph $\mathcal{G} = (V, E_c, E_p)$, we create a part clique $C_{P_i} \subseteq \mathcal{G}$ representing this part by taking all the slots $\mathbf{u}_{ij} \in V$ associated with this part, as well as all the part edges $\mathbf{e}_{ijk} \in E_p$. We remove the all contact edges $\mathbf{e}_{ij} \in E_c$ that connects P_i to other parts in the shape. Our goal, then, is find a set of part cliques $\mathbf{C}$ that can be connected together into a novel slot graph $\mathcal{G}' = (V', E_c', E_p')$, where $V' = \{\mathbf{u} \in \mathbf{C}\}$, $E_p' = \{\mathbf{e} \in \mathbf{C}\}$, and E_c' is the set of contact edges that need to be added to make all the slots attached i.e. connected to exactly one other slot via a contact edge.

There can be thousands of parts in available shape datasets, each containing multiple slots that can be attached in different ways. Thus, it is infeasible to search this combinatorial space exhaustively. Instead, we *learn* how to build novel slot graphs autoregressively, attaching one part clique at a time to a partial slot graph, until it is complete (i.e. all slots are attached). We learn this autoregressive process with teacher forcing: for every training sample, we take a random, connected partial slot graph $\mathcal{G}'$ consisting of one or more part cliques from a graph $\mathcal{G} = (V, E_c, E_P)$ extracted from a dataset shape S. We then select a random part clique $C_{P_j} | P_j \in S$ (referred to as C_{target} in the following sections) that is attached to $\mathcal{G}'$ on one or more slots $V_{\text{target}} = \{\mathbf{u}_{ij} \mid \mathbf{u}_{ij} \in \mathcal{G}', \mathbf{u}_{ji} \in C_{P_j}\}$ via set of contact edges $E_{\text{target}} = \{\mathbf{e}_{ij}^c \mid \mathbf{u}_{ij} \in V_{\text{target}}\}$. The goal of a single generation step, then, is to maximize

$$p(V_{\text{target}}, C_{\text{target}}, E_{\text{target}} \mid \mathcal{G}')$$

Rather than learn this complex joint distribution directly, we instead factor it into three steps using the chain rule:

- **Where** to attach: maximizing $p(V_{\text{target}} \mid \mathcal{G}')$
- **What** to attach: maximizing $p(C_{\text{target}} \mid \mathcal{G}', V_{\text{target}})$
- **How** to attach: maximizing $p(E_{\text{target}} \mid \mathcal{G}', V_{\text{target}}, C_{\text{target}})$

In the remainder of this section, we detail the formulation for the networks we use for each of the three steps, as well as how we use them during test time. Figure 7.2 visually illustrates these steps.

Where to Attach?: Given a partial slot graph $\mathcal{G}'$, we first identify the slots V_{target} to which the next-retrieved part clique should attach. We predict each element of V_{target} autoregressively (in any order), where each step i takes as input $\mathcal{G}'$ and the already-sampled slots $V^i_{\text{target}} = \{V_{\text{target},0} \dots V_{\text{target},i-1}\}$ (highlighted in $\mathcal{G}'$ with a one-hot node feature). We first use a MLP f_{continue} to predict the probability p_{continue} that another slot should be added ($p_{\text{continue}} = 0$ if $V^i_{\text{target}} = V_{\text{target}}$ and 1 otherwise). If more slots should be included, then we use an MLP f_{next} to predict a logit for each of the unattached slots $\tilde{V}$ in $\mathcal{G}'$ that are not already in V^i_{target}. These logits are then fed into a softmax to obtain a probability distribution over possible slots:

$$p(V_{\text{target}}|\mathcal{G}') = \prod_{i=1}^{|V_{\text{target}}|} p^i_{\text{cont}} \cdot p^i_{\text{next}}[V_{\text{target},i}]$$

$$p^i_{\text{cont}} = \begin{cases} p_{\text{continue}}(h_{\mathcal{G}'}|V^i_{\text{target}}) & i < |V_{\text{target}}| \\ 1 - p_{\text{continue}}(h_{\mathcal{G}'}|V^i_{\text{target}}) & i = |V_{\text{target}}| \end{cases}$$

$$p^i_{\text{next}} = \text{softmax}(\left[f_{\text{next}}(h_{\mathbf{u}}|V^i_{\text{target}})|\mathbf{u} \in \tilde{V}/V^i_{\text{target}}\right])$$

What to Attach?: Having selected the slots V_{target} to attach to, we then retrieve part cliques compatible with the partial graph $\mathcal{G}'$ and the selected slots. Similar to prior work [117], we take a contrastive learning approach to this problem: the probability of the ground truth part clique should be greater than that of randomly sampled other part cliques (i.e. negative examples) by some margin m.

$$p(C_{\text{target}} \mid \mathcal{G}', V_{\text{target}}) > p(C_{\text{negative}} \mid \mathcal{G}', V_{\text{target}}) + m$$

We use two neural networks to enforce this property. The first maps part cliques C into an embedding space $\mathbb{R}_{\text{emb}}$.

$$X_C = f_{\text{emb}}(h_C)$$

where f_{emb} is the embedding MLP and h_C is the graph feature computed from C alone. The second network is a mixture density network (MDN) that outputs a probability distribution over this embedding space:

$$P(X|\mathcal{G}', V_{\text{target}}, X \in \mathbb{R}_{\text{emb}}) = \text{MDN}(h_{\mathcal{G}'}, h_{\mathcal{G}'_{\text{target}}})$$

Where V_{target} are highlighted in the input node features and $h_{\mathcal{G}'_{\text{target}}}$ is obtained by computing graph features using V_{target} only. More precisely, it represents the conditional probability distribution $P(X|\mathcal{G}', V_{\text{target}}, X \in \mathbb{R}_{\text{emb}})$ as a mixture of N gaussians, with mixing coefficients $\pi_1 \dots \pi_N$, means $\mu_1 \dots \mu_N$ and standard deviations $\sigma_1 \dots \sigma_N$ respectively. The probability of any embedding X_C, then, can be expressed as

$$p(X_C) = \sum_{k=1}^{N} \pi_k \cdot \mathcal{N}(X_C \mid \mu_k, \sigma_k^2)$$

We omit the conditions $(\mathcal{G}', V_{\text{target}})$ for simplicity of notation. In practice, we use negative log likelihood to setup the triplet loss:

$$\ell(X_C) = -\log \sum_{k=1}^{N} \pi_k \cdot \mathcal{N}(X_C \mid \mu_k, \sigma_k^2)$$

Given a positive example C_{target} and a negative example C_{negative}, we then obtain the final triplet loss as

$$\mathcal{L}(X_{C_{\text{target}}}, X_{C_{\text{negative}}}) = \max\{m + \ell(X_{C_{\text{target}}}) - \ell(X_{C_{\text{negative}}}), 0\}$$

Where m is a constant margin. We select the negative examples C_{negative} at training time by computing the triplet loss between the positive example and a set of randomly sampled negative examples, and choose one that gives a non-zero loss, whenever possible (i.e. using only semi-hard triplets).

We visualize the behavior of this module trained on Chair in Figure 7.3. When the input demands a very specific type of structure (first 2 rows), our module can retrieve the part cliques that match such structure. When the input has fewer constraints (3rd row), our module retrieves a wide variety of partial cliques that can be attached. In the 4th row, our module retrieves chair legs that are *structurally* compatible. The legs are not necessarily *geometrically* compatible, as geometry information is not available to the module.

How to Attach?: The last module learns to connect the retrieved part clique C_{target} to the partial slot graph $\mathcal{G}'$. It predicts a probability for every pair of slots $\mathbf{u}_{ij} \in V_{\text{target}}, \mathbf{u}_{ji} \in C_{\text{target}}$ that could be connected via a contact edge:

$$p(\mathbf{e}_{ij}^c \mid \mathcal{G}, V_{\text{target}}, C_{\text{target}}) = f_{\text{edge}}(h_{\mathbf{u}_{ij}}, h'_{\mathbf{u}_{ji}})$$

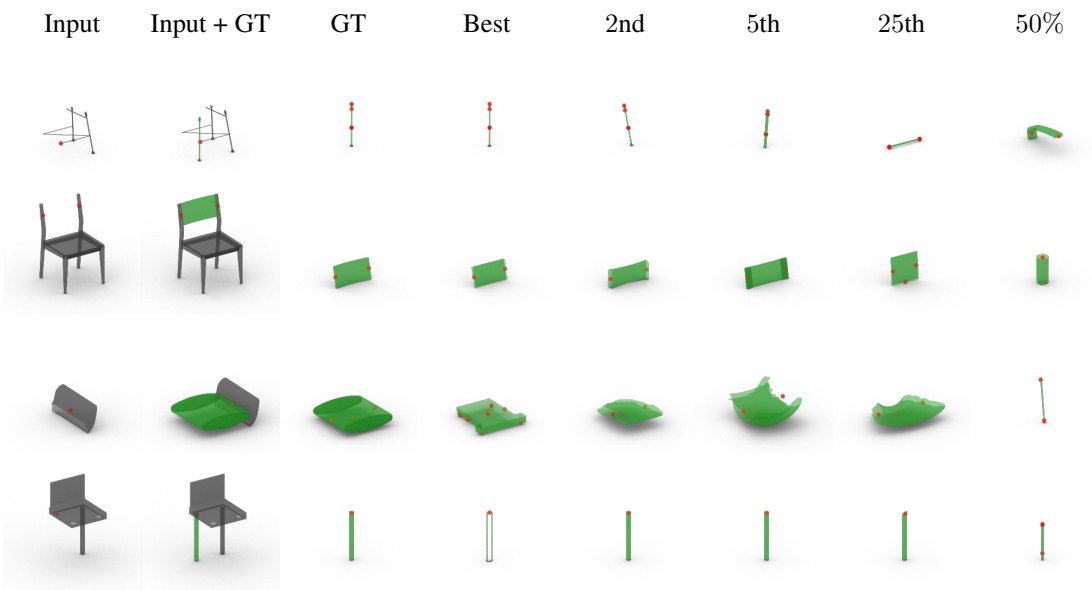


Figure 7.3: Example outputs of the **What to Attach?** module. We visualize the input partial slot graph within the parts that contain them (grey) and the center of the selected slots (red), as well as the ground truth part (green, 2nd column). The parts and slots are in their ground truth world-space pose, which is *not* available to the neural network. We then visualize, individually, the ground truth part and the retrieved candidates ranked 1st, 2nd, 5th, 25th, and at the 50th percentile, respectively, along with all of their slots (red). e

Where V_{target} are highlighted in input node features, f_{edge} is a MLP and $h_{\mathbf{u}_{ij}}$ and $h'_{\mathbf{u}_{ji}}$ are computed with two neural networks, one over $\mathcal{G}'$ and one over $\mathcal{C}_{\text{target}}$. $p(\mathbf{e}^c_{ij}) = 1$ if $\mathbf{e}^c_{ij} \in E_{\text{target}}$ and 0 otherwise. If both V_{target} and C_{target} contain one slot, then these slots must be connected, and this module can be skipped. To encourage the networks to learn more general representations, we augment V_{target} with random unattached slots in $\mathcal{G}'$.

Generating New Slot Graphs at Test Time: At test time, we generate new slot graphs by iteratively querying the three modules defined above. Although the modules we learn are probabilistic and support random sampling, we find MAP inference sufficient to produce a diverse range of shapes. Although trained on all shapes with less than or equal to 30 parts, less than 5 percent of the training shapes have more than 20 parts, and each of those shapes have rather unique structures. Therefore, when generating new slot graphs, we only use parts from shapes with less than 20 parts. We start each shape by randomly selecting a part from the candidate parts. We then iteratively query the three neural network modules, until the slot graph is complete (when all slots are attached). During this generation process, we use the output of the three neural modules to detect and reject partial slot graphs that are outliers:

- If the **Where** module gives a probability p_{continue} of less than 0.5 when there are no slots selected.
- Not all part cliques retrieved by the What module are good candidates. We reject a retrieved candidate

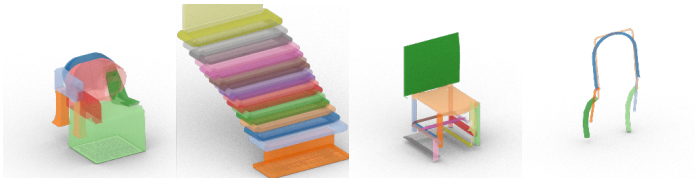


Figure 7.4: Typical structural outliers detected at test time. From left to right: redundant component (chair back), repetition of structures, inability to resolve local connections (chair base), not enough slots to finish structure.

C_{target} if $|V_{\text{target}}| > |C_{\text{target}}|$, or if one of the edge predicted by the **Where** module has a probability less than $1/(\max(|V_{\text{target}}| + 1, |C_{\text{target}}|) + 0.5)$. If all candidates within a margin of 60 (100 for parts invovled in symmetry) from the highest scoring candidate are rejected, we reject the partial slot graph.

We also reject the generated slot graph if any of the following conditions are met:

- The slot graph contains more than 20 parts.
- The slot graph is detected as an outlier. We perform outlier detection using one-class Support Vector Machines (OCSVM). We fit one OCSVM for all graphs in the training set with the same number of parts. For each OCSVM that fits graphs with N nodes, we use a feature size of $3(N-1)$, with the following features:
  - Number of parts with an (adjacency) degree of $1 \ldots N-1$.
  - Number of parts where $1 \ldots N-1$ other parts are within a distance of 2.
  - Number of parts where the furthest part has a distance of $1 \ldots N-1$.

  Other commonly used graph summary statistics, such as clustering coefficient, number of n-cycles, etc. are also possible candidates here, but we found the set of features we use to be sufficient for our purposes.

Finally, we also include logic to enforce part symmetries. When retrieving a part to connect with slots that were part of a symmetry group in their original shape, we alter the rank order of parts returned by our "what" module to prioritize (a) parts that occurred in symmetry groups in the dataset, followed by (b) parts that are approximately symmetrical according to chamfer distance.

7.4 Assembling Shapes From Slot Graphs

A generated slot graph defines connectivity between shape parts but does not explicitly give part poses. Thus, our system has an additional step to find world-space poses for all the retrieved parts. In this section, we

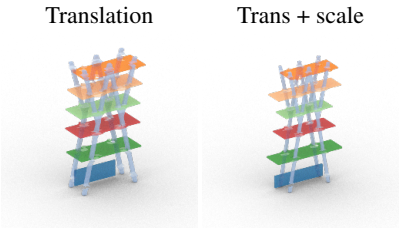


Figure 7.5: Optimizing part affine transformations to satisfy a slot graph. We show the output of the initial translation-only phase of optimization & the final output with both translation and scale.

describe a simple gradient-descent-based optimization procedure for doing so, which takes a generated slot graph $\mathcal{G} = (V, E_c, E_p)$ that describes N parts $P_1 \dots P_N$, and predicts an affine transformation matrix T_i for each part P_i.

Objective Function: To assemble a shape from its slot graph, we want each slot to be connected in the same way as it was in its original shape, which we approximate by enforcing that the distance from any point on a slot to the contacting slot should stay the same in the assembled shape. Formally, for each slot $\mathbf{u}_{ij} \in V$, we select the set of points S_{ij} that the slot contains (from the point sample of P_i from Section 7.2). For each point $p \in S_{ij}$, we compute its distance $d_o(p)$ to the closest point on the slot that was originally connected to $\mathbf{u}_{ij}$ in the dataset. We then optimize for $T_1 \dots T_N$ via the following objective: for every slot $\mathbf{u}_{ij} \in V$, every point sample $p \in S_{ij}$ should have the same distance to the connecting slot $\mathbf{u}_{ji}$ as the original distance $d_o(p)$:

$$f(T_1 \dots T_N) = \sum_{\mathbf{u}_{ij} \in V} \sum_{p \in S_{ij}} \Big(\big(\min_{q \in S_{ji}} d(T_i p, T_j q) \big) - d_o(p) \Big)^2$$

Optimization Process: We minimize this objective using gradient descent. Rather than full affine transformations, we optimize only translation and anisotropic scale. This prevents certain part re-uses from happening (e.g. re-using a horizontal crossbar as a vertical crossbar), but we find that the space of possible outputs is nonetheless expressive. To minimize unnecessary distortion, we prefer translation over scaling whenever possible: we optimize for translation only for the first 1000 iterations, and then alternate between translation and scaling every 50 iterations for the next 500 iterations. Optimizing for scales is essential in cases where translation alone cannot satisfy the slot graph. We show one such example in Figure 7.5, where the shelf layers are scaled horizontally to match the V shape of the frame.

Table 7.1: Comparing our system to baselines and ablations on generating visually and physically plausible shapes.

Category	Method	Root↑	Stab↑	Fool↑	FD↓	Parts
Chair	Ours	98.1	70.1	6.4	61.1	7.8
	ComplementMe	90.2	41.1	6.6	83.0	5.7
	StructureNet	81.0	61.3	4.0	37.5	12.1
	Oracle	94.5	83.4	25.8	13.3	–
	ComplementMe (w/sym)	88.3	79.1	21.9	21.6	4.3
	Ground Truth	100.0	100.0	–	–	11.1
	Ours (no symmetry)	98.2	68.0	8.1	58.9	7.8
	Ours (no duplicate)	97.5	67.4	13.8	61.2	7.7
Table	Ours	98.2	82.8	10.6	61.6	6.8
	ComplementMe	90.2	62.0	7.7	93.5	4.7
	StructureNet	82.8	78.5	2.3	85.2	7.8
	ComplementMe (w/sym)	87.1	84.0	35.8	18.7	3.3
	Ground Truth	100.0	100.0	–	–	9.3
Storage	Ours	99.4	90.8	15.5	42.6	6.9
	StructureNet	89.6	82.2	6.8	105.5	8.3
	ComplementMe (w/sym)	85.3	70.6	11.5	71.4	3.3
	Ground Truth	100.0	99.4	–	–	13.6
Lamp	Ours	89.6	–	21.5	42.4	3.4
	ComplementMe	62.0	–	35.8	26.0	3.4
	Ground Truth	92.6	–	–	–	4.2

7.5 Results & Evaluation

In this section, we evaluate our method's ability to synthesize novel shapes.

Implementation Details: We use the PartNet [82] dataset, segmenting shapes with the finest-grained PartNet hierarchy level and filtering out shapes with inconsistent segmentations and/or disconnected parts. Appendix A.7 provide more details on the data preparation process. We set the rounds of message passing, T, to 10 for all our graph neural networks (GNN) operating on partial slot graphs. We set $T = 4$ for GNNs operating on part cliques. Since no adjacency edges exists, this effectively leads to 2 rounds of message passing. We set the dimension of node embeddings to 64 and the dimension of graph embeddings to 128. All MLP we use have 2 hidden layers and uses leaky ReLU as the activation function. We use a mixture of 10 gaussians for the MDN and a margin $m = 20$ for the triplet loss. We train all neural networks with the Adam [61] optimizer, and with a batch size of 32. We select the negative examples for the **What** module from 32 randomly selected slot graphs as well, for each training step.

Novel Shape Synthesis: Figure 7.6 shows examples of shapes our method is capable of assembling. Our model learns to generate shapes with complex structures specific to each shape categories. Though it does not use global part position information during graph generation, the resulting slot graphs lead to consistent

Figure 7.6: Examples of range of shapes our method is able to generate. Each part has a different color.

global positions for the individual parts once optimized. Although we choose not to encode full geometries, our model is still able to generate shapes with interesting variations both structurally and geometrically. We further quantitatively compare the results generated by our model against against these alternatives:

- **ComplementMe** [117] is the previous state-of-art for modeling by part assembly. It retrieves compatible parts and places them together by predicting per-part translation vectors. ComplementMe also does not implement a stopping criteria for generation, so we train a network that takes a partial shape point cloud as input and predicts whether generation should stop. We also stop the generation early if the output of the part retrieval network does not change from one step to the next. Finally, ComplementMe relies on a part discovery process where most groups of symmetrical parts are treated as a single part

Figure 7.7: (a): Chairs in the first row of Figure 7.6, where parts coming from the same source shape now have the same color. (b): Geometric nearest neighbor of the the same chairs in the training set. (c): Chairs generated without enforcing part symmetries. (d): Chairs generated with the explicit enforcement that no parts coming from the same source shape can be attached to each other. Our method uses parts from different shapes to generate novel shapes. It can generate approximately symmetric shapes without explicit rules, and can also connect parts from different shapes together plausibly.

,

(e.g. four chair legs). We notice that, when trained on our data, ComplementMe suffers from a significant performance decrease on Chair and Table, and does not work on Storage at all (See Table 7.1). Therefore, for these categories, we also include results where parts are grouped by symmetry (w/sym) for reference. We stress that, under this condition, both retrieving and assembling parts are significantly easier, thus the results are not directly comparable.

- **StructureNet** [81] is an end-to-end generative model outputs a hierarchical shape structure, where each leaf node contains a latent code that can either be decoded into a cuboid or a point cloud. We modify it to output meshes by, for each leaf node, retrieving the part in the dataset whose StructureNet latent code is closest to the leaf node's latent code and then transforming the part to fit the cuboid for that leaf node.
- We also include an **Oracle** as an upper bound on retrieval-based shape generation. The oracle is an autoregressive model that takes as input at each step (a) the bounding boxes for all parts in a ground-truth shape and (b) point clouds for parts retrieved so far. Retrieved parts are scaled so that they fit exactly to the bounding box to which they are assigned.

See Appendix A.8 for more details about these baselines. We use an evaluation protocol similar to ShapeAssembly [56] which evaluates both the physical plausibility and quality of generated shapes:

- **Rootedness ↑ (Root)** measures the percentage of shapes for which there is a connected path between the ground to all parts;
- **Stability ↑ (Stable)** measures the percentage of shapes that remains upright under gravity and a small force in physical simulation;
- **Realism ↑ (Fool)** is the percentage of test set shapes classified as "generated" by a PointNet trained to distinguish between dataset and generated shapes;
- **Freschet Distance ↓ (FD)** [49] measures distributional similarity between generated and dataset shapes in the feature space of a pre-trained PointNet classifier.
- **Parts** is the mean number of parts in generated shapes.

Table 7.1 summarizes the results. By using a contact-based representation, our model is able to generate shapes that are more physically plausible (rooted and stable) than the baselines. while being comparable in terms of the overall shape quality, measured by Frechet distance and classifier fool percentage. Our model performs particularly well for storage furniture; we hypothesize rich connectivity information of this shape category allows our model to pick parts that match particularly well. Our model fares less well on lamps, where connectivity structure is simple and the geometric variability of parts (which out model does not encode) is highly variable. ComplementMe works well on lamps, thanks to its focus on part geometry. Its performance drops significantly on all other categories with more complicated shape structures. We provide more details, as well as random samples for all methods, in Appendix B.7.

Generalization Capacity: It is important that a generative model that follows the modeling by assembly paradigm learns to recombine parts from different sources into novel shapes. We demonstrate our model's capacity for this in Figure 7.7: it is able to assemble parts from multiple source shapes together into novel shapes different from those seen during training, with or without explicit restrictions whether parts from the same source shape can be connected to each other. We also see that while including symmetry reasoning improves geometric quality, our method is able to generate shapes that are roughly symmetrical without it. This is also reflected in Table 7.1: removing symmetry or prohibiting using multiple parts from the same source shape has minimal impact on our metrics. We provide more analysis of generalization in Appendix B.7.

Performance of Individual Modules: Finally, we evaluate each individual model module, using the following metrics:

- **Attach Acc**: How often the "where" module correctly selects the slots to attach, given the first slot.

Table 7.2: Evaluating our neural network modules in isolation.

	Attach Acc	Avg Rank	Edge Acc
Chair	96.70	99.05	94.79
Table	92.32	99.14	92.82
Storage	87.46	99.08	85.38
Lamp	98.87	91.36	91.89

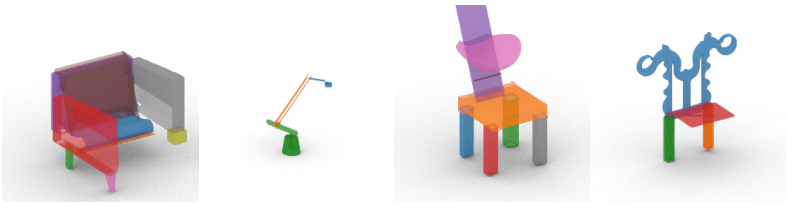


Figure 7.8: Typical failure cases of our method. From left to right: a chair with a tiny seat, two opposite-facing lamps attached together awkwardly, a chair with a implausible back, a chair that misses seat and legs completely.

- **Average Rank**: Average percentile rank of ground truth next part according to the "what" module.
- **Edge Acc**: How often the "how" module recovers the the ground truth edge pairs.

Table 7.2 summarizes the results. Modules perform very well, with some lower numbers caused by inherent multimodality of the data.

Limitations: Even with outlier detection as mentioned in section 5.3, poor-quality outputs can still occur. Figure 7.8 shows typical examples. Most are caused by our model's lack of focus on geometry: chairs with a tiny seat, lamps that face opposite directions, and chair backs that block the seat completely. Incorporating additional geometric features when appropriate could help.

7.6 Conclusion

In this chapter, we presented the Shape Part Slot machine, a new modeling-by-part-assembly generative model for 3D shapes. Our model synthesizes new shapes by generating slot graphs describing the contact structure between parts; it then assembles its retrieved parts by optimizing per-part affine transforms to be consistent with this structure. The slot graph encodes surprisingly little information, yet we demonstrated experimentally that our model outperforms multiple baselines and prior modeling-by-assembly systems on generating novel shapes from PartNet parts.

There are multiple directions for future work. Parts could be repurposed in more diverse ways if we had a method to transfer slot graphs between geometrically- and contextually-similar parts (so e.g. a chair seat

that had armests originally does not have to have them in all synthesized results). More variety could also be obtained by optimizing for part orientations (so e.g. a vertical slat could be used as a horizontal one).

Chapter 8

Conclusions and Future Directions

In this dissertation, we have proposed a series of autoregressive generative models of 3D shapes and scenes. Our key contributions have addressed the central questions raised in the introductory chapter, enabling generation of 3D shapes and scenes with better quality, complexity, diversity, and are easier to interact with. We formulated autoregressive generation by proposing novel input representations, effective output factorizations, optimization strategies to combine components together, and techniques to improve compatibility between individual components. We enhanced the interpretability and controllability by introducing a two-level paradigm that separates high-level semantics from low-level details. We encouraged our models to learn generalizable rules by selecting appropriate presentations and training techniques. We conducted extensive qualitative and quantitative evaluations, and demonstrated the effectiveness of our proposed methods in creating 3D shapes and scenes with better quality, complexity, diversity and controllability.

8.1 Future Work

Although this dissertation takes an important step towards learning autoregressive generative models of 3D shapes and scenes, there exist numerous opportunities for future works. We have already discussed various immediate opportunities for extending our method in the summaries of the respective chapters: more principled approaches for extracting relation graphs, encoding more functional relationships, improving the deformation of 3D rooms, repurporsing shape parts in more diverse ways, etc. In this section, we put the work of this dissertation in a broader context and discuss other opportunities for future work.

8.1.1 Extending the Scope of 3D Shapes and Scenes

While dissertation focuses on 3D shapes and scenes of indoor environments, the techniques used in this dissertation could be extended to shapes and scenes in other domains, such as building facades, street layouts, CAD models, procedural models, and in general, any domain that involves some element of *design*. Large-scale datasets for many of such domains [110, 109, 129, 64] have become available in recent years, while it is also worth trying to collect and build new dataset for other domains. In this dissertation, many heuristics and simplifications were used as the datasets we employed lack the appropriate information to support more principled and complex approaches. For example, if furniture assembly instructions were available, it would be possible to use a more physically accurate definition of "slots", and remove many of the relationship extraction heuristics describe in Section A.7. Subsequently, an interesting direction would be to collect new datasets with the demand of generative models in mind. We have demonstrated that additional module are needed when extending generative models from room-level scenes to floor level scenes and shapes e.g. the need to deform components, the use of metric learning to select compatible components, etc. When extending our approaches to other domains, it will also be likely that additional modifications are necessary. It will be crucial to survey existing approaches in these domains, and identify the key elements that need to be added to an autoregressive generative model in that domain. For example, existing methods for generating larger scale layout often have a strong reliance on optimization [146] and simulation [33]. How to incorporate these into an autoregressive generative model is an interesting direction.

8.1.2 More Flexible Source of Data

This dissertation relies on datasets of 3D shapes and scenes where each of them are represented with individual components with 3D geometry. Such data requires effort to obtain: someone needs to model the shapes and scenes in 3D and compile them into a dataset. In contrast, information about 3D shapes and scenes also exist in many other forms of data that are much easier to obtain, such as photos, videos and text descriptions. There exist not only much larger datasets in these mediums [72, 153, 95], but also large "foundation" models [10] that are trained on these datasets and can be adapted to a wide range of tasks. Finding a way to leverage these data sources not only gives training data that a several orders of magnitude larger in size, but also gives access to information about shapes and scenes that are unlikely to have a corresponding 3D model e.g. historical objects that are only documented in paintings and books. Recent works have already begun using such strategies, for example, leveraging foundation models of images [94] to guide the generation of

3D shapes [53, 91]. The sequential nature of autoregressive generative models poses challenges for utilizing mediums such as images, in that structured information needs to be extracted to enable sequential generation. On the other hand, there are unique opportunities for utilizing other mediums that more closely parallel an autoregressive generation process. For example, text specifications of a shape / scene can be more easily processed by an autoregressive model. Proper incorporation of such sources could be an effective way to further improve the performance of our autoregressive generative models.

8.1.3 Learning Design Principles

Earlier rule based methods suffer from extensive need for human effort to manually craft the rules. This dissertation addresses this issue by automatically learning generative models from large datasets. However, rule-based methods and learning based methods are not orthogonal. As stated in the introductory section, one of the main goal of this dissertation is encouraging generative models to learn rules that can generalize well. We achieved this with a variety of techniques. However, all these techniques are rather ad-hoc: none of them strive to represent the design principles that underlies the creation of 3D shapes and scenes. Designing an autoregressive model that recovers the actual design principles that underlies the creation of 3D shapes and scenes is an important future direction. To achieve this, adopting more neurosymbolic approaches [98] might be necessary, in which one learns to generate symbolic programs that are then executed to create the final shapes and scenes. The "plan and instantiate" approach introduced in Chapter 5 exhibits neurosymbolic characteristics to some degree: the high level "plan" can be seen as a constraint program that are then executed into the final scene through "instantiation". However, the learned object placement priors used in the instantiation process are not neurosymbolic. Instead of use such priors, one can instead learn a model that generates a symbolic program describing how an object should be placed, which, when executed, yields location distributions like those in Figure 5.9. It would also be helpful to use datasets which not only contains the final shapes and scenes, but also the processes in which they are created. For examples, datasets of computer-aided design (CAD) models [109, 129] may be more suitable for this purpose, as they provide information about the process in which the shapes are designed. For scenes, one could imaging collecting a dataset of building information modeling (BIM) models, which contain more comprehensive information about the physical and functional characteristics of indoor spaces.

8.1.4 Better and More Comprehensive Evaluation Metrics

Throughout the dissertation, we have provided visualizations of the output distributions of the individual decisions steps e.g. Figure 3.5, 4.3, 5.10, 6.3, 7.3, etc. We discussed how these outputs are desirable, and alluded how they can be useful for editing tasks. However, there is a lack of a systematic method to evaluate the accuracy of the outputs. One of the main reasons for this is the absence of ground truth to compare against. Taking the location module in Figure 4.3 as an example: there is no ground truth data for "where to place a double bed", as the dataset we use only give information for a single possible location per given room. Instead of directly evaluating the quality of such distributions, we resorted to evaluation of the final generated shapes and scenes, showing that we are able to generate outputs of better quality and diversity. However, quality and diversity of final output does not necessary correlate to the accuracy of the individual decision steps, as combinatorics could play an important role here: even if every decision step involves only a binary choice, through many decision steps, a model can still easily create a huge diversity of outputs. Improving the accuracy of these single decisions steps could further enhance the quality and diversity of the final outputs. More importantly, doing so would make the models easier to interact with: a user might want to get interactive design suggestions on where to place a decoration, a service robot might need to hallucinate all the potential locations of a dining table, data augmentation pipelines might need to create multiple small variations of a shape or a scene. In all these scenarios, it is important that we accurately model all the single step distributions. A metric that directly evaluates such accuracy would be helpful here. Since no ground truth are available in existing datasets, it will be necessary to either collect new data or to create synthetic datasets from scene grammar where such ground truth are readily available.

Moreover, even when evaluating the "final" outputs, the metrics used by the dissertation thus far are often too one-dimensional. For example, we used Freschet distance and real-fake classification accuracy to measure overall distributional similarity between generated and ground truth samples. However, it is uncertain what elements of the output contributes the most to these metrics. For example, in an earlier experiment, we noticed that position and existence of beds has a much larger impact on these metrics compared to smaller objects, such as nightstands. Using a single number to judge the overall quality of output, subsequently, could lead to bias, focusing on only making sure the larger objects are placed correctly. A more systematic study of these metrics would be helpful in avoiding such biases. Development of new and better metrics is also an option here. One option is to conduct large-scale perceptual studies on human perception of 3D shapes scenes, and use the result of such studies to create a feature space that correlates better with human judgment of shape

and scene quality, similarity, and correctness.

8.1.5 Application to Downstream Tasks

Last but not least, it is worth revisiting one of the main motivations behind this dissertation: the increasing demand for virtual 3D indoor environments, especially for training for downstream tasks in vision and robotics. In this dissertation, we have demonstrated that our generative models can generate a large amount of new synthetic data. However, there is still much to be done to apply these models to downstream tasks. In Chapter 6, we briefly explored using the output of our floor generative model as training data for virtual navigation agents. However, we only showed that agents trained on the generated scenes can navigate reasonably well in unseen environments. What we weren't able to show is that addition of generated synthetic data can improve the performance of such agents. Much work is yet to be done to make our generative models an actual effective data source for such downstream tasks. Besides obvious options such as improving the quality and diversity of the generated outputs, it is also worth bench-marking the aspects of the generated data that have the most impact on downstream task performances, exploring the possibility of using controllable generative models to create on-demand data that target samples where the models struggle to learn, or even designing a framework where one jointly learn a 3D data generator and a model that uses such data.

Another possibility is to leverage the prior knowledge learned by our generative models. For example, they can be used to complete invisible regions of shapes and scenes in tasks such a single-view reconstruction. They can also be used to identify and correct predictions that are clearly out of distribution. Compared to fixed simplifying assumptions, such as the Manhattan world assumption [24], rules learned by generative models can provide a more flexible set of constraints for tasks such as scene layout estimation. Robotics agents often need to navigate to an unknown location that contains a specified object, or to interact with a new object; a generative model could serve as an effective guide in these scenarios. More generally, 3D generative models allow artificial intelligence agents to build abstract internal models of the 3D world, enabling better and more generalizable interpretation of visual observations. How to concretely connect our generative models with these tasks remains an interesting open direction.

Appendix A

Additional Implementation Details

A.1 Dataset Filtering (Chapter 3)

We apply the following filters to SUNCG scenes to produce our training/test datasets:

Low-quality rooms: Each SUNCG scene has been quality-rated by three human raters. We discard all rooms that do not have the highest rating, as well as any remaining rooms with large object interpenetrations.

Aggregate objects: We remove rooms containing objects that are actually sets of multiple objects, e.g. *chair_set* and *double_desk*.

Architectural features: We filter out rooms containing objects that are actually architectural features which alter the room geometry, such as *partition*, *column*, *stairs* and *arch*.

Inhabitants: We remove objects representing people and pets.

Infrequent objects: We filter out rooms containing unusual, infrequently occuring object categories. Specifically, we filter for categories that occur in less than 5% of rooms and are not functionally important, e.g. *vacuum_cleaner*, *fish_tank*, *cart*, *tripod*.

Outlier room sizes: We are interested in modeling residential-scale rooms, and our image-based scene representation must fit the extents of all training rooms. Thus, we filter out rooms larger than $6\,\mathrm{m} \times 6\,\mathrm{m}$ in floor plan size and taller than $4\,\mathrm{m}$. We also filter outliers in terms of object density by removing rooms with fewer than 4 or greater than 20 objects.

Rugs: We remove floor rugs, as they are very frequently non-uniformly (and thus non-physically) scaled to a large range of sizes and aspect ratios (up to the size of an entire room), making them inappropriate for category-based analysis.

A.2 Baseline Object Arrangement Model (Chapter 3)

Existing object arrangement methods differ in terms of input/output assumptions, datasets, and user interactivity, making fair comparison hard. We select data-driven model elements common to multiple approaches to construct a representative baseline.

We learn our object arrangement baseline model by extracting pairwise observations for all objects in each room type. The pairwise observations in the training set are used to estimate parameters for three types of priors:

- Relative 2D offset between closest points on object o_i and point on reference object o_r in coordinate frame of o_r.
- Relative 2D clearance of object o_i from closest point on a wall in coordinate frame of o_i, and angle between front vector of o_i and offset to closest point on wall in frame of o_i.
- Relative orientation of front vector of o_i relative to front vector of o_r in coordinate frame of o_r.

Parameter estimation

The 2D offsets and 2D clearances between objects, and between an object and a wall point are used to estimate the parameters of a Gaussian mixture model $p(\theta) = \sum_{i=1}^{K} \phi_i \mathcal{N}(\mu_i, \Sigma_i)$. We use variational Bayesian estimation to infer the weights ϕ_i of up to $K = 5$ components, with a tied covariance matrix between components, and a Dirichlet process concentration prior set to $1/K$. The angles characterizing relative front orientation between objects are used to estimate the PDF over the angular domain as a histogram discretized into 16 equal bins of width $\pi/8$.

Synthesis procedure

At synthesis time, we order all objects to be arranged by decreasing bounding box volume. For each object, we uniformly sample 2D positions on the floor of the room, and assign one of eight cardinal orientations at random. The sampled position and orientation is checked for collisions with other objects or room wall geometry. If collisions exist, we take a new random sample.

We sample $n = 2000$ times and take up to $m = 200$ non-colliding candidate placements. We evaluate the probability of each candidate placement by treating the object to be placed as a reference object and computing the pairwise offset and orientation of all present objects and the closest point on a wall. The pairwise offset, clearance from wall, and relative orientations are evaluated against all pairwise priors to

obtain the relative offset probability $p_{\text{o}}(o_i, o_r)$, relative front angle probability $p_{\text{a}}(o_i, o_r)$, and wall clearance and orientation probability $p_{\text{w}}(o_i, p_w)$ where p_w is a point on a room wall. The overall log probability of a candidate placement x is then:

$$\log(p(x)) = p_{\text{cooc}}(w_{\text{o}} \log(p_{\text{o}}(x)) + w_{\text{a}} \log(p_{\text{a}}(x)) + w_{\text{w}} \log(p_{\text{w}}(x)))$$

where p_{cooc} is the co-occurrence probability of the object category pair, estimated from the ratio of observations involving the pair out of all pairwise observations in the training set. We sum this pairwise log probability over all pairs involving the object to be placed, to estimate the likelihood of the candidate placement. We then take the placement with the maximum probability, and proceed to the next object in order of decreasing size. Arrangement ends when the smallest object has been placed. As this synthesis procedure has no mechanism for automatically rejecting questionable scenes, we disable that feature in our system in comparisons between the two.

A.3 Model Architecture Details (Chapter 4)

Here we give specific details about the neural network architectures used for each of our system's modules.

A.3.1 Next Category

The module uses a Resnet18 [47] to encode the scene image. It also extract the counts of all categories of objects in the scene (i.e. a "bag of categories" representation), as in our prior work [125], and encodes this with a fully-connected network. Finally, the model concatenates these two encodings and feeds them through another fully-connected network to output a probability distribution over categories. At test time, the module samples from the predicted distribution to select the next category. Figure A.1 shows the architecture diagram for this network.

A.3.2 Location

Figure A.2 shows the architecture diagram for this module. It uses a Resnet34 [47] to encode the scene image. It is followed by five "up-convolutional" (i.e. transpose convolution) blocks (*UpConvBlock*). Up-convolution is done by first nearest-neighbor upsampling the input with scale factor of 2, and then applying a 3x3 convolution. Finally, we apply a 1x1 convolution to generate a $(C + 1) \times 64 \times 64$ distribution over

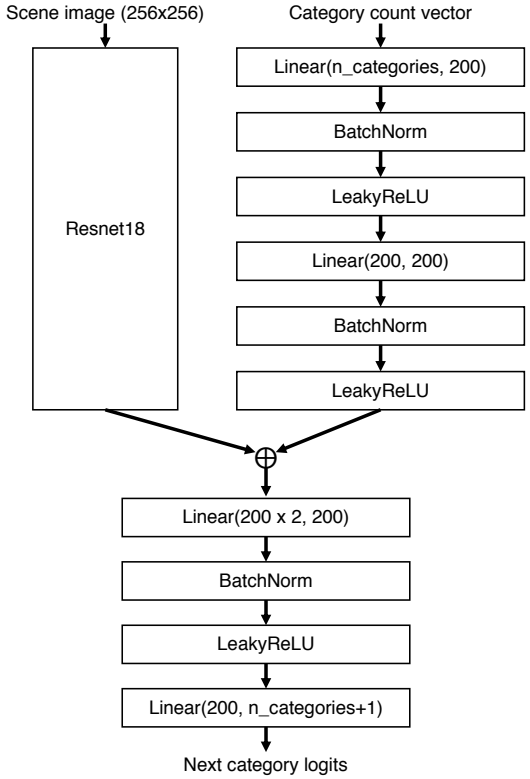


Figure A.1: Architecture diagram for the next category prediction module.

categories and location, where C is the number of categories for the room type.

Since the target output during the training process (exact location of the object centroids for the room) is different from the outcome we prefer (a smooth distribution over all possible locations), the module has a high potential to overfit. To alleviate this, we apply dropout before and after the Resnet34 encoder, and also before the final 1x1 convolution. We also apply L2 regularization in the training process. We found this combination of techniques effective at preventing overfitting, though we have not quantitatively evaluated the behavior of each individual component.

A.3.3 Orientation

Given a translated top-down scene image and object category, the orientation module predicts what direction an object of that category should face if placed at the center of the image. Figure A.3 shows the architecture diagram for this module. We assume each category has a canonical front-facing direction. Rather than predict the angle of rotation θ, which is circular, we instead predict the front direction vector, i.e. $[\cos\theta, \sin\theta]$. This must be a normalized vector, i.e. the magnitude of $\sin\theta$ must be $\sqrt{1-\cos^2\theta}$. Thus, our module predicts

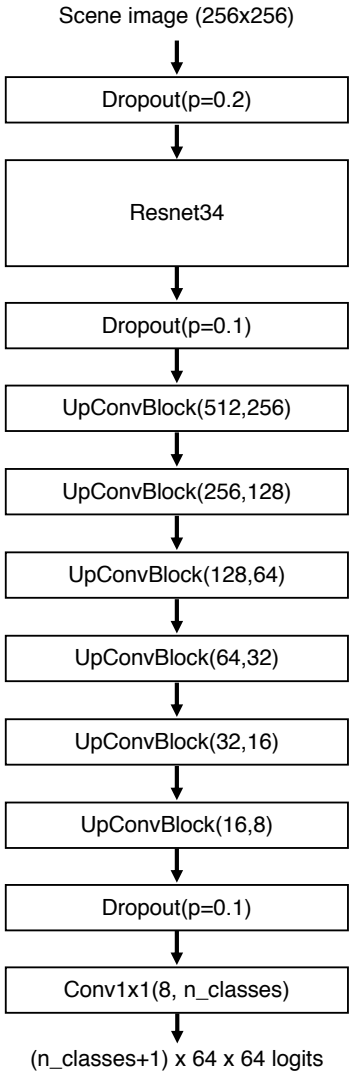


Figure A.2: Architecture diagram for the location prediction module. An *UpConvBlock* is a 3x3 transpose convolution with stride 2 followed by a Batch Normalization layer and a ReLU layer.

$\cos\theta$ along with a Boolean value giving the sign of $\sin\theta$ (more precisely, it predicts the probability that $\sin\theta$ is positive). Here, we found using separate network weights per category to be most effective.

The set of possible orientations has the potential to be multimodal: for instance, a bed in the corner of a room may be backed up against either wall of the corner. To allow our module to model this behavior, we implement it with a conditional variational autoencoder (CVAE) [113]. Specifically, we use a CNN to encode the input scene (the *Conditional Prior*), which we then concatenate with a latent code z sampled from a multivariate unit normal distribution, and then feed to a fully-connected *Decoder* to produce $\cos\theta$ and the sign of $\sin\theta$. At training time, we use the standard CVAE loss formulation to learn an approximate posterior

distribution over latent codes).

Since interior scenes are frequently enclosed by rectilinear architecture, objects in them are often precisely aligned to cardinal directions. A CVAE, however, being a probabilistic model, samples noisy directions. To allow our module to produce precise alignments when appropriate, this module includes a second CNN (the *Snap Predictor*) which takes the input scene and predicts whether the object to be inserted should have its predicted orientation "snapped" to the nearest of the four cardinal directions.

A.3.4 Dimensions

Given a scene image transformed into the local coordinate frame of a particular object category, the dimensions module predicts the spatial extent of the object. That is, it predicts an object-space bounding box for the object to be inserted. This is also a multimodal problem, even more so than orientation (e.g. many wardrobes of varying lengths can fit against the same wall). Again, we use a CVAE for this: a CNN encodes the scene, concatenates it with z, and then uses a fully-connected decoder to produce the $[x, y]$ dimensions of the bounding box. Figure A.4 shows the architecture diagram for this module.

The human eye is very sensitive to errors in size, e.g. a too-large object that penetrates the wall next to it. To fine-tune the prediction results, we include an adversarial loss term in the CVAE training. This loss uses a convolutional *Discriminator* which takes the input scene concatenated channel-wise with the signed distance field (SDF) of the predicted bounding box. As with orientation, this module also uses separate network weights per category.

A.4 Dataset Details (Chapter 4)

We adopt similar dataset filtering strategies as that in Chapter 3, with a few notable differences:

1. We manually selected a list of frequently-occurring objects, which we allow to appear on top of other objects (only on the visible top surface, i.e. no televisions contained in a TV stand). We remove all second tier objects whose parents were filtered out.

2. To facilitate matching objects by bounding box dimensions, we discard rooms containing objects which are scaled by more than 10% along any dimensions. For objects scaled by less than that, we remove the scaling from their transformation matrices.

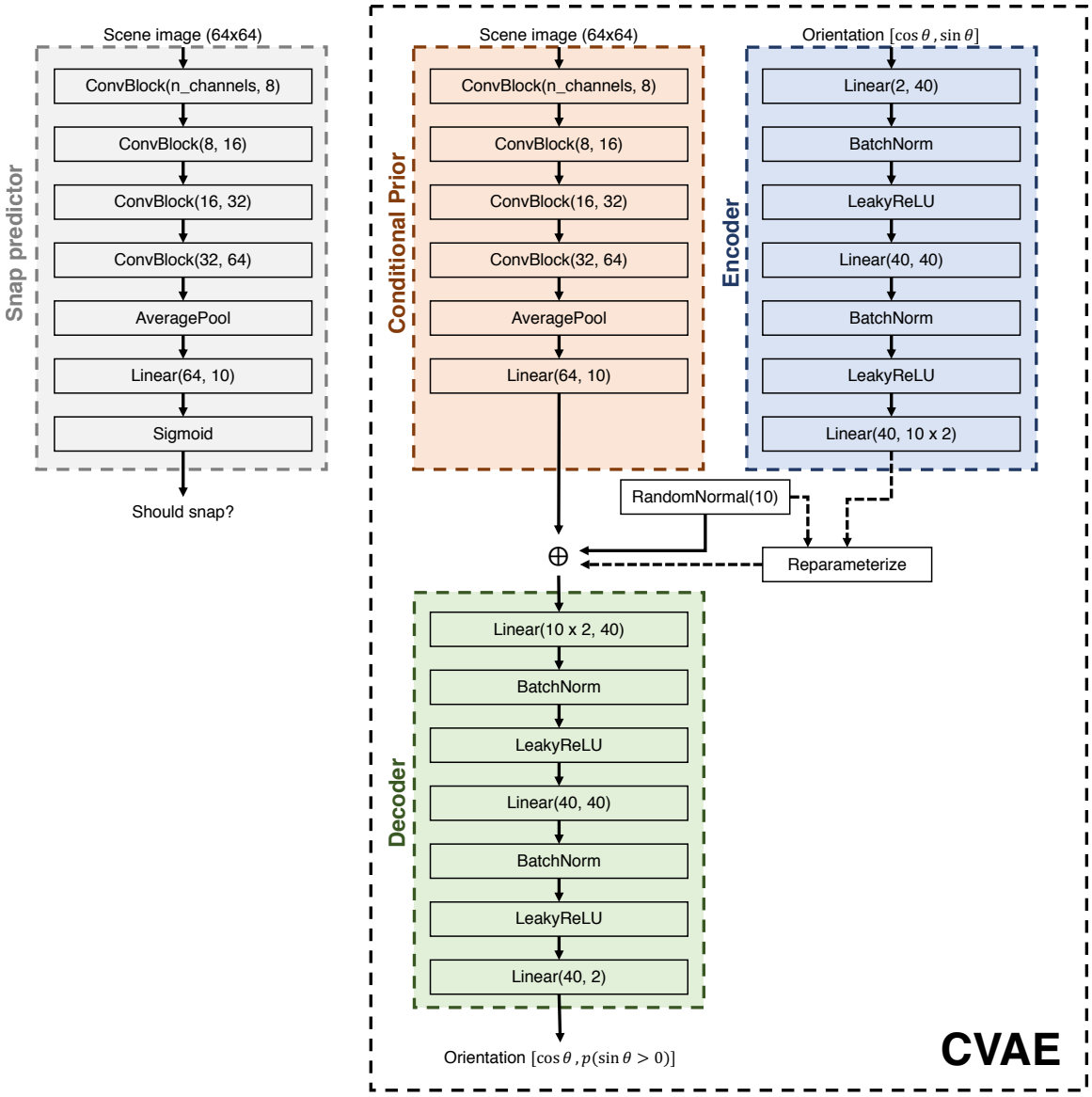


Figure A.3: Architecture diagram for the orientation prediction module. A *ConvBlock* is a 3x3 convolution with stride 2 followed by a Batch Normalization layer and a ReLU layer.

3. We augment the living room and office dataset with 4 different rotations $(0°, 90°, 180°, 270°)$ of the same room during training, to reduce overfitting, particularly for the location module.

Table A.1 shows the counts of all categories appearing in the four types of rooms used in this work, where possible second tier categories are highlighted with bold. The categories are arranged in the order that they are given to the category module.

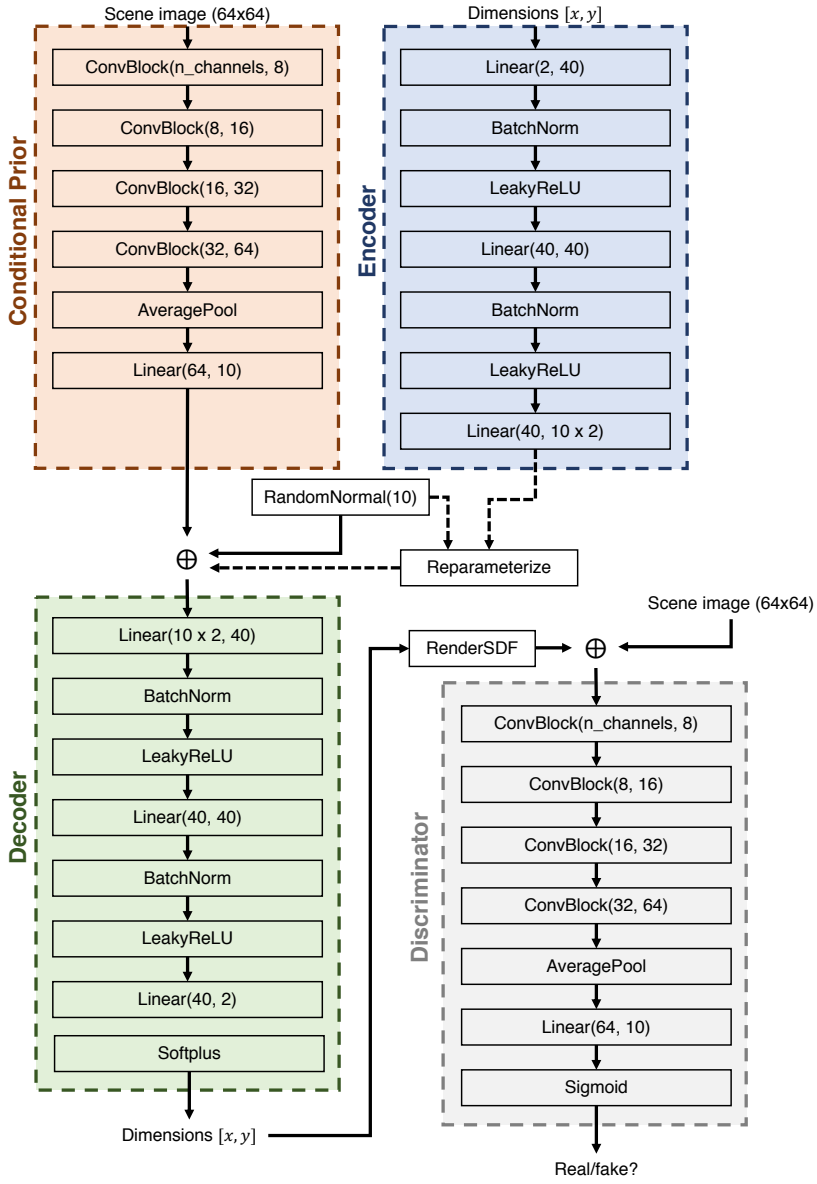


Figure A.4: Architecture diagram for the dimensions prediction module. A *ConvBlock* is a 3x3 convolution with stride 2 followed by a Batch Normalization layer and a ReLU layer.

Bedroom									
Name	door	window	double bed	wardrobe	single bed	desk	stand	bunker bed	dresser
Count	8335	7104	4230	6690	2032	2586	6508	593	2222
Name	tv stand	**plant**	sofa	dressing table	office chair	coffee table	sofa chair	ottoman	shelving
Count	1432	1341	393	1730	2077	1048	727	1142	901
Name	floor lamp	armchair	daybed	**table lamp**	baby bed	piano	shoes cabinet	straight chair	hanger
Count	1387	703	181	2844	266	102	619	595	540
Name	**television**	bench chair	**toy**	**laptop**	loudspeaker	chair	**book**	**console**	whiteboard
Count	1252	134	231	1010	438	98	858	368	105
Name	stool	pedestal fan	**vase**	**fishbowl**					
Count	178	320	395	59					
Living Room									
Name	door	window	sofa	sofa chair	coffee table	**plant**	tv stand	floor lamp	fireplace
Count	2286	1789	1661	983	1336	1059	696	651	257
Name	piano	wardrobe	ottoman	shelving	armchair	loudspeaker	dresser	**television**	**toy**
Count	85	172	314	309	204	384	64	447	56
Name	stand	**vase**	**console**	straight chair	shoes cabinet	bench chair	stool	hanger	**laptop**
Count	101	282	187	51	31	26	33	34	72
Name	**table lamp**	pedestal fan	**fishbowl**	**book**	**cup**	**fruit bowl**	**glass**	**bottle**	
Count	68	40	21	69	59	17	16	15	
Office									
Name	door	window	desk	sofa	office chair	**plant**	shelving	sofa chair	wardrobe
Count	1572	1307	1616	313	1577	557	764	290	319
Name	armchair	piano	tv stand	coffee table	floor lamp	straight chair	dresser	bench chair	ottoman
Count	300	75	137	137	256	274	84	37	112
Name	fireplace	whiteboard	**laptop**	stand	**toy**	**table lamp**	**book**	loudspeaker	shoes cabinet
Count	38	100	377 102		52	294	356	113	39
Name	hanger	stool	**television**	**vase**	pedestal fan	water machine	**console**	**fishbowl**	**cup**
Count	51	56	84	132	49	32	30	5	27
Bathroom									
Name	door	window	bathtub	shower	sink	toilet	shelving	**plant**	washer
Count	7448	4299	5441	5308	4627	6437	3963	1045	1428
Name	bidet	wardrobe	stand	dresser	floor lamp	ottoman	cabinet	hanger	trash can
Count	1753	459	315	89	231	95	57	111	409
Name	**toy**	coffee table	straight chair	**vase**					
Count	126	31	20	31					

Table A.1: Counts for all the object categories appearing in the four types of rooms used in Chapter 4, in the order that they are presented to the category prediction module. Bold category name indicates that this can be a second tier object

A.5 Graph Extraction Heuristics (Chapter 5)

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.5.1 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.5.2 Edge Pruning

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

Kept edges: We always retain support edges, wall $\rightarrow$ 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 $\rightarrow$ object edges that are not *adjacent*, to reduce the overwhelmingly large number of such edges (44 – 59% of all '$\rightarrow$ object' edges in the initial graphs, depending on room type). There are also many occurrences of 'double edges,' i.e. both the edges $A \rightarrow B$ and $A \leftarrow B$ 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 $A \rightarrow B$ 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.5.3 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 $\rightarrow$ 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).

A.6 Backtracking Details (Chapter 5)

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 5.4.1, 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 5.2. 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.

A.7 Data Preparation (Chapter 7)

We use the PartNet [82] dataset for all our experiments, following the train/validation/test split provided in the original paper.

Table A.2: Dataset statistics before and after our filtering process

Category	Split	Before	After
Chair	Train	4489	3315
	Val	617	438
	Test	1217	886
Table	Train	5707	4254
	Val	843	637
	Test	1668	1257
Storage	Train	1588	1123
	Val	230	152
	Test	451	290
Lamp	Train	1554	1187
	Val	234	181
	Test	419	321

A.7.1 Obtaining Part Level Geometry

Each shape in the PartNet data comes with a semantic hierarchy that decomposes the shape into parts in a coarse-to-fine manner. We use the finest-grained level of parts in this hierarchy. We filter the data using the following criteria:

- We remove shapes that contain only 1 part or more than 30 parts.
- We detect inconsistent shapes with parts that do not equal the union of their children. For the chairs and tables dataset, we reject these inconsistent shapes. The furniture and lamps dataset are smaller, so in these datasets, we keep the inconsistent parts, but discard their children.
- We remove shapes with parts that contain floating geometry due to annotation errors. We detect such cases by first clustering the part's point cloud with DBSCAN [32]. If there exist any cluster that is significantly smaller than other clusters, we reject the entire shape. (We cannot reject all parts that consist of multiple disconnected clusters, because some parts contain multiple symmetric disconnected components).
- We remove shapes that are disconnected based on the adjacency edges we detect.

Table A.2 summarizes the size of the dataset before and after filtering.

A.7.2 Extracting Relationships Between Parts

After obtaining the geometry of individual parts, we sample the surface of each part uniformly to obtain a 3000-point representation. We then detect relationships between parts based on the protocol of StructureNet [81]:

Detecting Symmetry: We detect symmetry based on the methods proposed by Wang et al. [127]. We restrict the symmetry types to translational symmetry, reflectional symmetry about planes parallel to the three coordinate planes, and 4-way rotational symmetry about the y(up)-axis. We create an undirected graph for each of the symmetry type, where every edge is a detected symmetry between a pair of parts. We treat each connected component in these graphs as a symmetry group.

Pruning Symmetry: We then prune the detected symmetries by enforcing that each part belongs to at most one symmetry groups. We prioritize larger groups. If two groups are of the same size, then we favor the simpler explanation: translational $>$ rotational $>$ reflectional.

Detecting Adjacency: We regard two parts, A and B, as adjacent if the smallest distance between their respective points clouds is less than $\tau = \theta \mathbf{r}$, where $\mathbf{r}$ is the average bounding sphere radius of the two parts. We first detect symmetries using $\theta = 0.05$ i.e. the orignal setting of StructureNet. We then do a second pass of adjacency detection for parts involved in symmetry groups in order to recover any undetected adjancies to a common neighbor: We set $\theta = 0.1$ if A belongs to a symmetry group (before pruning) and B is adjacent (using $\theta = 0.05$) to any other parts in the same symmetry group, vice versa; we further increase θ to 0.3 if the involved symmetry group has more than 3 parts and at least 3 other parts are adjacent to B, vice versa. We use the same threshold τ for computing the points for each slot (Section 4.2).

Pruning Adjacency: We then attempt to identify the set of adjancency relationships that best describe the part structure. Note that this might not be necessary for a dataset where the connections between parts are more clearly defined. We prune the adjacency edges using the following set of heuristics, applied in order. All heuristics are only applied if removing the edge does not disconnect the adjacency graph:

We first remove any edges between parts in the same symmetry group, prior to symmetry pruning.

We then identify all triplet of parts A, B, C that overlap at the same area, and thus pairwise adjacent. For each triplet, we check if there's an edge that we can prune, using the following heuristics, without loss of generality, applied in order:

- If B and C shares a common parent in the PartNet hierarchy and A has a different parent, then we store either AB or AC for deletion if we can break ties between them: we store the edge for the part that is either significantly farther from A, smaller in surface area, or with less adjacent parts. We do not store any edges if the ties between B and C cannot be broken.
- We store AC for deletion if the y(up)-coordinate of the centroid of B is between those of A and C, and is of at least a distance of 0.05 away from each.
- We store BC for deletion if the surface area of both part B and C is signifcantly smaller than that of

part A, or if B and C has roughly the same area but A does not.

We then sort all the candidate edges for deletion, prioritizing on those detected with heuristics mentioned earlier, and then those belonging to parts with smaller surface areas.

After finding all the candidates edges, we iterate over them and delete edges, while respecting the detected symmetries. For each candidate AB, we check if A and/or B belongs to any symmetry groups. If A is in a symmetry group, then we include all other adjacency edges from any parts in that symmetry group to B. We do the same for B. We proceed to remove all these edges if the following conditions are met:

- Removing these edges does not disconnect the graph.
- Removing these edges does not disconnect a symmetry group from its most frequent neighbor i.e. the part that has the highest number of adjacency edges to parts in the symmetry group. If multiple such neighbors exist, we prefer to keep the edges to the neighbor that is not in any symmetry groups. If there are still multiple such neighbors, we keep only 1 of them, and allow deleting edges to the rest. A special case occurs when every neighbor to a symmetry group is adjacent to exactly one part in the group. This often occurs when a group of symmetrical parts are decomposed further into subparts (e.g. four symmetrical legs are decomposed into four legs and for leg wheels). In such cases, we regard every part in the adjacent symmetry group as a most frequent neighbor.
- Either A and B are still both connected to C in the original triplet, or if there exists other parts in the region where A, B and C overlaps and a path can be found from A to B via those parts or vice versa.

A.8 Details on Baselines (Chapter 7)

We provide additional details on how we implemented the baselines.

ComplementMe: We re-implemented ComplentMe [117] in PyTorch. We mostly used the original settings of ComplementMe, with the following exceptions:

- We set the maximum threshold for the standard deviation of the Gaussian Mixture model to 50 instead of 0.05, since we found that the standard deviation of all Gaussians saturate at the original threshold very quickly.
- ComplementMe sampled random triplets originally, we instead sample only the semi-hard triplets i.e. triplets that give a non-zero training loss.
- In the paper, ComplementMe suggests that the placement networks do not share weights with the retrieval/embedding networks. This is not the case in their official implementation. We followed the

description in the paper.

- We removed all BatchNorm layers from the PointNet backbone since we observed that including them hurts the evaluation performance.

We train ComplementMe until convergence.

StructureNet: We use the pre-trained models provided by StructureNet [81], which are trained on the same split as what we use in the paper. Do note that StructureNet uses a different data filtering strategy than ours, so the training set will differ slightly. We encode every part in the test set using the pre-trained part encoder, with each part centered and normalized in the same way as they would be if used to train StructureNet. We then use the provided evaluation script to randomly sample outputs. Instead of decoding child-level latent code to point clouds, we directly retrieve the test set part that is the closest in the latent space, and then apply the predicted transformations to the retrived part.

Appendix B

Additional Results and Evaluations

B.1 Random Scene Samples (Chapter 3-5)

We include additional results from our generative models of room-level scenes (Chapter 3-5). Figure B.1 shows random samples from Deep Synth (Chapter 3). Figure B.2, B.3 shows random samples from Fast Synth (Chapter 4). Figure B.4, B.5 shows random samples from PlanIT (Chapter 5).

Figure B.1: Random samples of 3D rooms generated by the method proposed in Chapter 3

Figure B.2: Random samples of 3D rooms generated by the method proposed in Chapter 4

Figure B.3: Random samples of 3D rooms generated by the method proposed in Chapter 4 (continued)

Figure B.4: Random samples of 3D rooms generated by the method proposed in Chapter 5

Figure B.5: Random samples of 3D rooms generated by the method proposed in Chapter 5 (continued)

B.2 Performance of Each Model Component (Chapter 4)

Table B.1 shows the performance of each of our modules on a held-out test set of scene data. Different metrics are reported for different modules, as appopriate. We have no natural baseline to which to compare these numbers. As an alternative, we report the improvement in performance relative to a randomly-initialized network.

Room Type	Cat (Top1)	Cat (Top5)	Loc (X-Ent)	Orient (ELBo)	Orient-Snap (Acc.)	Dims (ELBo)
Bedroom	0.5000 (+0.4225)	0.8650 (+0.7600)	0.0030 (-98.61%)	0.0899 (-54.20%)	0.8821 (+0.3821)	0.0018 (-97.74%)
Living	0.5375 (+0.5312)	0.8719 (+0.8001)	0.0035 (-98.56%)	0.1093 (-44.99%)	0.8902 (+0.3902)	0.0018 (-97.79%)
Office	0.5664 (+0.5525)	0.8948 (+0.7629)	0.0038 (-98.25%)	0.0639 (-68.03%)	0.9482 (+0.4482)	0.0015 (-98.14%)
Bathroom	0.6180 (+0.5873)	0.9573 (+0.8597)	0.0020 (-85.00%)	0.0906 (-53.23%)	0.9419 (+0.4419)	0.0019 (-97.64%)

Table B.1: Performance of each component of our model on held-out test data. *Acc.* is binary classification accuracy; *Top N* is top-n multiclass classification accuracy; *X-Ent* is cross-entropy; *ELBo* is the standard Evidence Lower Bound objective for variational autoencoders. The numbers in parentheses show the improvement relative to a randomly-initialized network.

B.3 Generalization (Chapter 4)

To evaluate if our models are merely "memorizing" the training scenes, we measure similar a generated room can be to a room in the training set. To do so, we use the same scene-to-scene similarity function as that in Figure 3.12 and compute the maximal similarity score of a generated room against $5,000$ rooms in the training set, We plot the score distribution for $1,000$ generated rooms in Figure B.6. For comparison, We also compute the same score for $1,000$ rooms from the training set (which are disjoint from the $5,000$ aforementioned rooms). In general, the behavior for the synthesized rooms is similar to that of scenes from the dataset. Our model definitively does not just memorize the training data, as it is actually less likely for our model to synthesize a room that is very similar to one from the training set. It is also less likely for our model to synthesize something that is very different from all other rooms in the training set. This is coherent with our impression: that our model suffers from minor mode collapses, and does not capture all possible unique room layouts. Finally, the large spike in extremely-similar rooms for the dataset-to-dataset comparison (the tall orange bar on the far right of the plot) is due to exact duplicate scenes with exist in the training set.

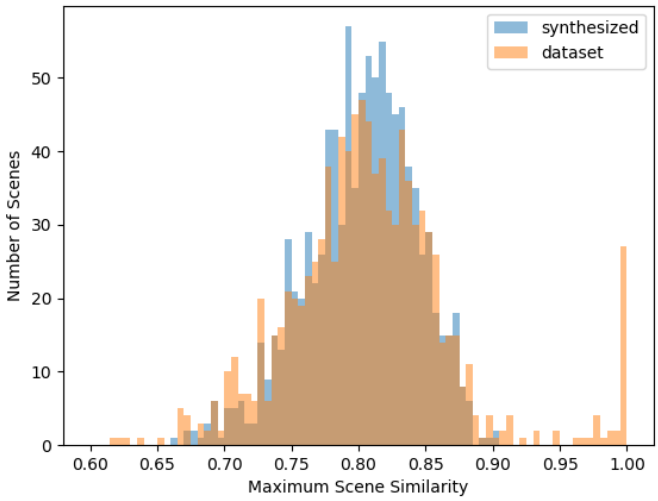


Figure B.6: Plotting the maximal similarity score of a bedroom against $5,000$ rooms from the training set. We plot the distribution of results for $1,000$ synthesized rooms and $1,000$ held out rooms in the training set (disjoint from the $5,000$)

.

B.4 Qualitative Results for the Navigation Experiment (Chapter 6)

Figure B.7 shows example trajectories of a pre-trained DDPPO [128] agent walking through scenes generated by our methods, as well an original Matterport3D [12] scene.

B.5 Walk-through of a Generated Scene (Chapter 6)

Figure B.8 shows a trajectory of a first-person walk through of one of the generated scenes. The original semantic annotation of Matterpot3D meshes are done on meshes reconstructed with a different pipeline, and subsequently of lower visual quality. To produce this trajectory, we manually annotated the higher quality meshes of the set of rooms contained in a layout generated by the smart portal stitching strategy, and then manually performed a walk through.

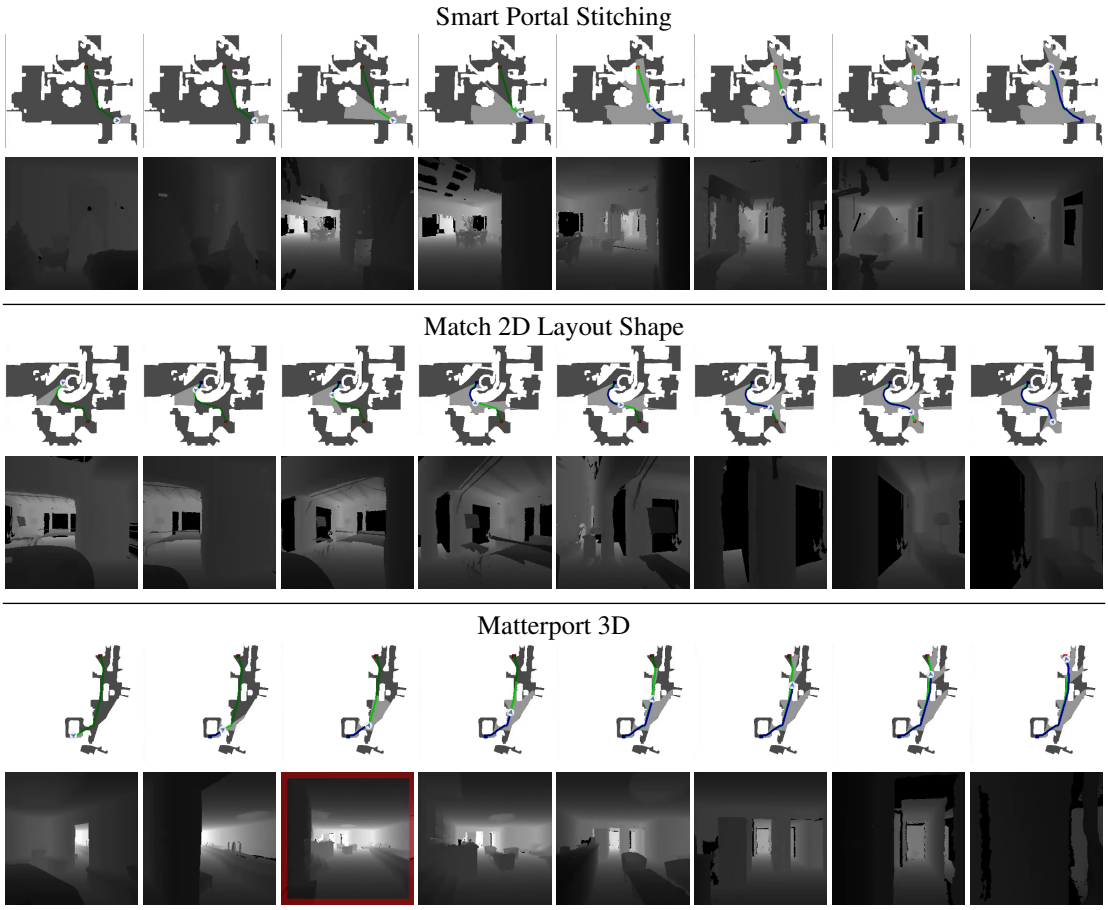


Figure B.7: Trajectory taken by a DDPPO agent through scenes generated by the proposed methods, as well as a scene taken directly from Matterport3D. Top row: top down view of the navigable areas. Bottom row: agent's first-person view of the scene, depth only.

B.6 Evaluating the Effect of Data Augmentation with Our Data (Chapter 6)

Directly evaluating the impact of data augmentation with our data is challenging, as it has been shown that navigation agents continue to learn from data after billions of steps [128]. Here, we provide an approximation by evaluating two agents that perform similarly on their respective training sets. We train one of the agent on 184 floor plans from the Matterport3D dataset, and the other the 184 Matterport3D floor plans, as well as 104 floor plans generated by the Smart Portal Stitching strategy. For the second agent, we construct the training set such that half of the episodes are from Matterport3D, and the other half from the generated data. We train the first agent for 30 million steps. We record the training success rate (about 0.67) and SPL (about

Figure B.8: Trajectory of a first person walk through for a scene generated by the smart portal stitching strategy.

Table B.2: Evaluating the impact of data augmentation with data generated by our methods. Higher values are better. Evaluation is done on the Gibson [136] dataset, in scenes unseen at training time.

Scene Source	**Success Rate**	**SPL**
Smart Portal Stitching + MP3D Scenes	0.831	0.662
MP3D Scenes	0.818	0.628

0.49), and then train the second agent until it reaches similar performances, at around 50 million steps. We then evaluate the trained agents on the Gibson validation set. The results are summarized in table B.2. The agent trained on Matterport3D + Smart Portal Stitching outperforms the agent trained on only Matterport3D with respect to both success rate and SPL. We do stress that this is only an approximation, and it is possible the better performance results from other factors. We leave rigorous, full-scale evaluation to future works.

B.7 More Results (Chapter 7)

We show random samples of our method and the baselines on all four categories in Figure B.9, B.10, B.11 and B.12. We also show random samples from the test set in Figure B.13. Overall, the quality and physical plausibility of the generated shapes correlate well with the quantitative metrics.

ComplementMe benefits considerably from grouping parts by symmetry, as it simplifies the task of predicting global poses of shapes significantly. When parts are not grouped by symmetry, it often fails to predict the right pose of parts, and sometimes is not able to complete a shape at all. It also produces a lot of incorrect

storages, even with the help of symmetry.

StructureNet is usually able to generate shapes that are plausible, though often with a few missing parts. However, it has the tendency to generate only a subset of shape types. This is most apparent for table and storage, when it generates mostly square tables and storages with open shelves. Large gaps sometimes exist between the individual parts, leading to problems with physical plausibility. Note that this problem is not caused by us retrieving parts directly using the latent code — the box version of StructureNet has similar issues (see the evaluation of ShapeAssembly [56]).

The behavior of our method is more polarizing: it generates a lot of high quality shapes; however, some other generated shapes are totally incorrect. The high quality shapes fare better than the baselines in terms of quality, physical plausibility, and diversity to some extent. The incorrect shapes exhibit a wide range of failure mode, which we hypothesize can be traced back to a few incorrect steps in the autoregressive generation process. Reducing the chance of these incorrect steps, and identifying them when they happen, is an important future direction to take in order to further improve the quality of the generated shapes. Our method also has the tendency to generate simpler shapes when sampling randomly. This is not caused by the neural networks learning biased distributions, but caused by the higher failure rate for more complex structure during autoregressive sampling. We also notice a few repeated shapes, especially for storages. This can be addressed by sampling the neural network modules randomly, as opposed to doing MAP inference.

Finally, we note that none of the methods are able to generate shapes that are close to the dataset in terms of quality and diversity. This is especially the case for shapes with unique and complex structures: they are harder to learn, and there is often not enough training data for them. Learning these structures correctly and efficiently remains an open problem.

Figure B.9: Chair Unconditional Samples

Figure B.10: Table Unconditional Samples

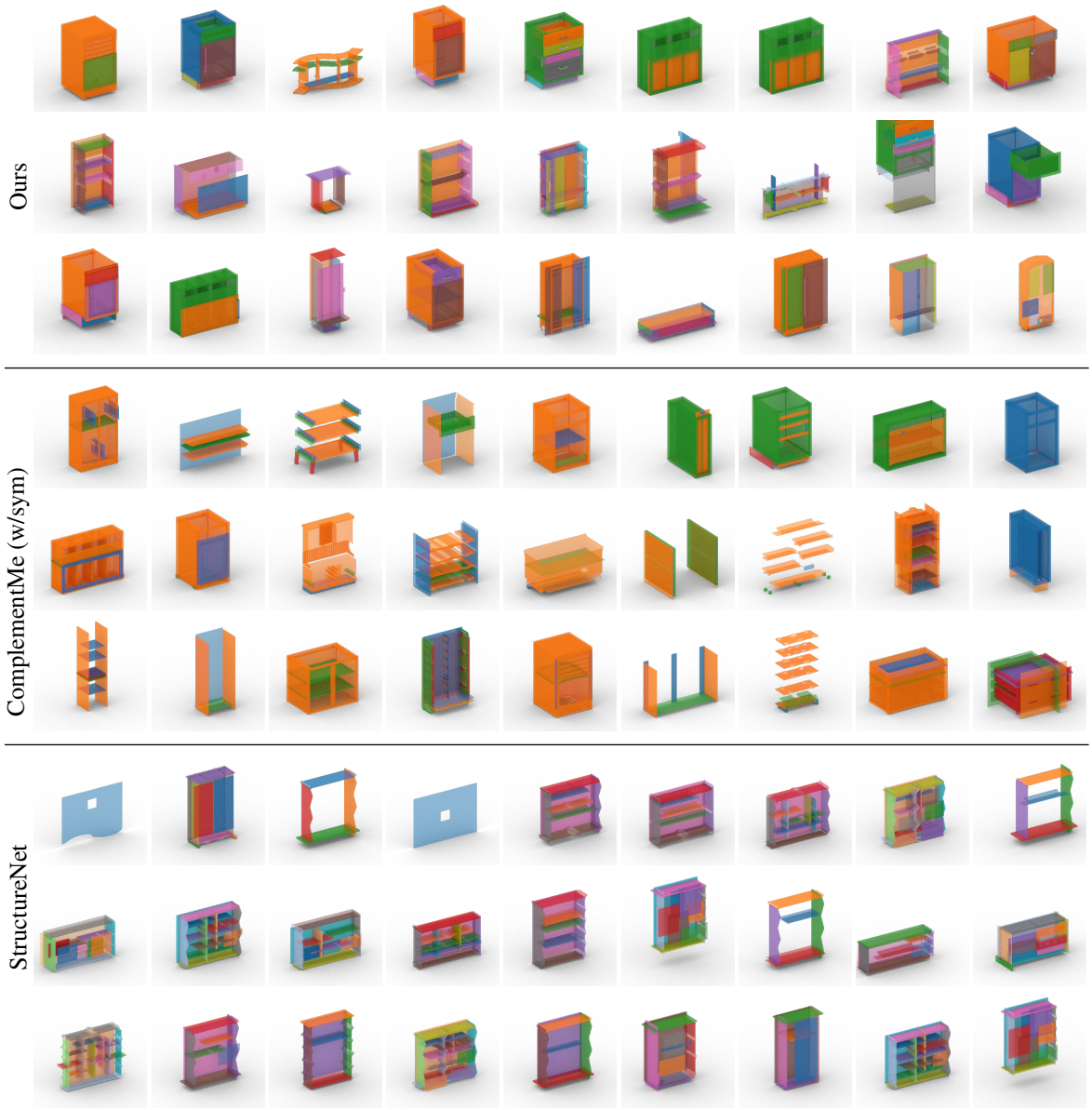


Figure B.11: Storage Unconditional Samples

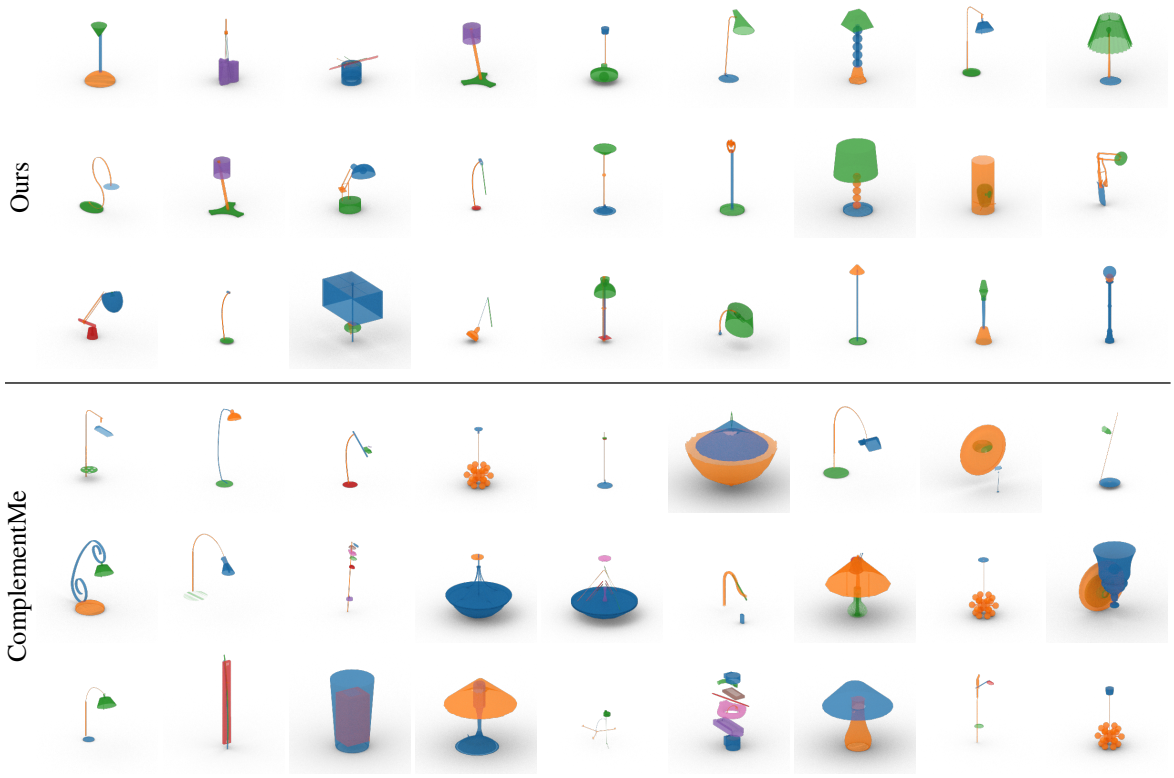


Figure B.12: Lamp Unconditional Samples

Figure B.13: Dataset Unconditional Samples

Bibliography

[1] Lifull home's dataset. https://www.nii.ac.jp/dsc/idr/en/lifull/, 2016. Accessed: 2016-11-2.

[2] Peter Anderson, Angel Chang, Devendra Singh Chaplot, Alexey Dosovitskiy, Saurabh Gupta, Vladlen Koltun, Jana Kosecka, Jitendra Malik, Roozbeh Mottaghi, Manolis Savva, and Amir R. Zamir. On evaluation of embodied navigation agents, 2018.

[3] Andrej Karpathy. char-rnn. https://github.com/karpathy/char-rnn, 2015. Accessed: 2018-01-20.

[4] Scott A Arvin and Donald H House. Modeling architectural design objectives in physically based space planning. *Automation in Construction*, 11(2):213–225, 2002.

[5] Melinos Averkiou, Vladimir Kim, Youyi Zheng, and Niloy J. Mitra. Shapesynth: Parameterizing model collections for coupled shape exploration and synthesis. *Computer Graphics Forum (Special issue of Eurographics 2014)*, 2014.

[6] Fan Bao, Dong-Ming Yan, Niloy J Mitra, and Peter Wonka. Generating and exploring good building layouts. *ACM Transactions on Graphics (TOG)*, 32(4):1–10, 2013.

[7] Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab, 2016.

[8] Christopher M Bishop. Mixture density networks. 1994.

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

[10] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[12] Angel Chang, Angela Dai, Thomas Funkhouser, Maciej Halber, Matthias Niessner, Manolis Savva, Shuran Song, Andy Zeng, and Yinda Zhang. Matterport3D: Learning from RGB-D data in indoor environments. 2017.

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

[14] Angel X Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, et al. Shapenet: An information-rich 3d model repository. *arXiv preprint arXiv:1512.03012*, 2015.

[15] Angel X Chang, Manolis Savva, and Christopher D Manning. Learning spatial knowledge for text to 3D scene generation. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[16] Chaos Group. 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, 2018. Accessed: 2018-10-13.

[17] Siddhartha Chaudhuri, Evangelos Kalogerakis, Stephen Giguere, and Thomas Funkhouser. AttribIt: Content creation with semantic attributes. *ACM Symposium on User Interface Software and Technology (UIST)*, Oct. 2013.

[18] Siddhartha Chaudhuri, Evangelos Kalogerakis, Leonidas Guibas, and Vladlen Koltun. Probabilistic reasoning for assembly-based 3d modeling. *ACM Trans. Graph.*, 30(4), July 2011.

[19] Siddhartha Chaudhuri, Daniel Ritchie, Jiajun Wu, Kai Xu, and Hao Zhang. Learning generative models of 3d structures. In *Computer Graphics Forum*, volume 39, pages 643–666. Wiley Online Library, 2020.

[20] Gal Chechik, Varun Sharma, Uri Shalit, and Samy Bengio. Large scale online learning of image similarity through ranking. *Journal of Machine Learning Research*, 11(Mar):1109–1135, 2010.

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

[22] Kang Chen, Kun Xu, Yizhou Yu, Tian-Yi Wang, and Shi-Min Hu. Magic Decorator: Automatic Material Suggestion for Indoor Digital Scenes. In *SIGGRAPH Asia 2015*, 2015.

[23] Zhiqin Chen and Hao Zhang. Learning implicit fields for generative shape modeling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5939–5948, 2019.

[24] James M Coughlan and Alan L Yuille. Manhattan world: Compass direction from a single image by bayesian inference. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 941–947. IEEE, 1999.

[25] Angela Dai, Angel X. Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. ScanNet: Richly-annotated 3D Reconstructions of Indoor Scenes. In *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 2017.

[26] Angela Dai, Daniel Ritchie, Martin Bokeloh, Scott Reed, Jürgen Sturm, and Matthias Nießner. Scancomplete: Large-scale scene completion and semantic segmentation for 3d scans. In *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 2018.

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

[28] B. Efron and R. Tibshirani. Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical Science*, 1(1):54–75, 2 1986.

[29] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. Learning to Infer Graphics Programs from Hand-Drawn Images. *CoRR*, arXiv:1707.09627, 2017.

[30] P. Erdos and A Renyi. On the evolution of random graphs. In *PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES*, pages 17–61, 1960.

[31] S. M. Ali Eslami, Nicolas Heess, Theophane Weber, Yuval Tassa, David Szepesvari, Koray Kavukcuoglu, and Geoffrey E. Hinton. Attend, Infer, Repeat: Fast Scene Understanding with Generative Models. In *NIPS 2016*, 2016.

[32] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

[33] Tian Feng, Lap-Fai Yu, Sai-Kit Yeung, KangKang Yin, and Kun Zhou. Crowd-driven mid-scale layout design. *ACM Trans. Graph.*, 35(4):132–1, 2016.

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

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

[36] Michael S. Floater. Mean value coordinates. *Comput. Aided Geom. Des.*, 20(1):19–27, Mar. 2003.

[37] Huan Fu, Bowen Cai, Lin Gao, Ling-Xiao Zhang, Jiaming Wang, Cao Li, Qixun Zeng, Chengyue Sun, Rongfei Jia, Binqiang Zhao, et al. 3d-front: 3d furnished rooms with layouts and semantics. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10933–10942, 2021.

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

[39] Thomas Funkhouser, Michael Kazhdan, Philip Shilane, Patrick Min, William Kiefer, Ayellet Tal, Szymon Rusinkiewicz, and David Dobkin. Modeling by example. In *ACM SIGGRAPH 2004 Papers*, 2004.

[40] Lin Gao, Jie Yang, Tong Wu, Yu-Jie Yuan, Hongbo Fu, Yu-Kun Lai, and Hao (Richard) Zhang. Sdm-net: Deep generative network for structured deformable mesh. In *SIGGRAPH Asia*, 2019.

[41] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. *CoRR*, arXiv:1704.01212, 2017.

[42] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Nets. In *NIPS 2014*, 2014.

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

[44] Karol Gregor, Ivo Danihelka, Alex Graves, and Daan Wierstra. DRAW: A recurrent neural network for image generation. In *ICML 2015*, 2015.

[45] Thibault Groueix, Matthew Fisher, Vladimir G Kim, Bryan C Russell, and Mathieu Aubry. A papier-mâché approach to learning 3d surface generation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 216–224, 2018.

[46] David Ha and Douglas Eck. A Neural Representation of Sketch Drawings. *CoRR*, arXiv:1704.03477, 2017.

[47] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR 2016*, 2016.

[48] Paul Henderson and Vittorio Ferrari. A Generative Model of 3D Object Layouts in Apartments. *CoRR*, arXiv:1711.10939, 2017.

[49] 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 *NeurIPS*, 2017.

[50] Ruizhen Hu, Zeyu Huang, Yuhan Tang, Oliver van Kaick, Hao Zhang, and Hui Huang. Graph2plan: Learning floorplan generation from layout graphs. *arXiv preprint arXiv:2004.13204*, 2020.

[51] R. Hu, M. Savva, and O. van Kaick. Functionality representations and applications for shape analysis. *Computer Graphics Forum*, 37(2):603–624, 2018.

[52] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-Image Translation with Conditional Adversarial Networks. In *CVPR 2017*, 2017.

[53] Ajay Jain, Ben Mildenhall, Jonathan T Barron, Pieter Abbeel, and Ben Poole. Zero-shot text-guided object generation with dream fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 867–876, 2022.

[54] Arjun Jain, Thorsten Thormählen, Tobias Ritschel, and Hans-Peter Seidel. Exploring shape variations by 3d-model decomposition and part-based recombination. *Comput. Graph. Forum*, 31(2pt3):631–640, May 2012.

[55] Wengong Jin, Regina Barzilay, and Tommi S. Jaakkola. Junction tree variational autoencoder for molecular graph generation. In *ICML 2018*, 2018.

[56] R. Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. Shapeassembly: Learning to generate programs for 3d shape structure synthesis. *ACM Transactions on Graphics (TOG), SIGGRAPH Asia 2020*, 39(6):Article 234, 2020.

[57] R. Kenny Jones, David Charatan, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. Shapemod: Macro operation discovery for 3d shape programs. In *SIGGRAPH 2021*, 2021.

[58] Arthur Juliani, Ahmed Khalifa, Vincent-Pierre Berges, Jonathan Harper, Ervin Teng, Hunter Henry, Adam Crespi, Julian Togelius, and Danny Lange. Obstacle tower: A generalization challenge in vision, control, and planning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2019.

[59] Evangelos Kalogerakis, Siddhartha Chaudhuri, Daphne Koller, and Vladlen Koltun. A probabilistic model for component-based shape synthesis. *ACM Trans. Graph.*, 31(4), July 2012.

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

[61] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *ICLR 2015*, 2015.

[62] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. In *ICLR 2014*, 2014.

[63] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR abs/ 1609.02907*, 2016.

[64] Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. Abc: A big cad model dataset for geometric deep learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9601–9611, 2019.

[65] Eric Kolve, Roozbeh Mottaghi, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: an interactive 3d environment for visual AI. *CoRR*, arXiv:1712.05474, 2017.

[66] Vladislav Kreavoy, Dan Julius, and Alla Sheffer. Model composition from interchangeable components. In *Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, PG '07, page 129–138, USA, 2007. IEEE Computer Society.

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

[68] Jun Li, Kai Xu, Siddhartha Chaudhuri, Ersin Yumer, Hao Zhang, and Leonidas Guibas. GRASS: Generative Recursive Autoencoders for Shape Structures. In *SIGGRAPH 2017*, 2017.

[69] Manyi Li, Akshay Gadi Patil, Kai Xu, Siddhartha Chaudhuri, Owais Khan, Ariel Shamir, Changhe Tu, Baoquan Chen, Daniel Cohen-Or, and Hao Zhang. Grains: Generative recursive autoencoders for indoor scenes. *CoRR*, arXiv:1807.09193, 2018.

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

[71] Yuan Liang, Song-Hai Zhang, and Ralph Robert Martin. Automatic data-driven room design generation. In Jiana Chang, Jian Jun Zhang, Nadia Magnenat Thalmann, Shi-Min Hu, Ruofeng Tong, and Wencheng Wang, editors, *Next Generation Computer Animation Techniques: Third International Workshop (AniNex 2017)*. 2017.

[72] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 740–755. Springer, 2014.

[73] Han Liu, Yong-Liang Yang, Sawsan Alhalawani, and Niloy J Mitra. Constraint-aware interior layout exploration for pre-cast concrete-based buildings. *The Visual Computer*, 29(6-8):663–673, 2013.

[74] Tianqiang Liu, Aaron Hertzmann, Wilmot Li, and Thomas Funkhouser. Style compatibility for 3D furniture models. In *SIGGRAPH 2015*, 2015.

[75] Andrew Luo, Zhoutong Zhang, Jiajun Wu, and Joshua B Tenenbaum. End-to-end optimization of scene layout. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3754–3763, 2020.

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

[77] Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-generated residential building layouts. In *ACM Transactions on Graphics (TOG)*, volume 29, page 181. ACM, 2010.

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

[79] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4460–4470, 2019.

[80] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. *CoRR*, arXiv:1301.3781, 2013.

[81] Kaichun Mo, Paul Guerrero, Li Yi, Hao Su, Peter Wonka, Niloy Mitra, and Leonidas Guibas. StructureNet: Hierarchical graph networks for 3D shape generation. In *SIGGRAPH Asia*, 2019.

[82] Kaichun Mo, Shilin Zhu, Angel X Chang, Li Yi, Subarna Tripathi, Leonidas J Guibas, and Hao Su. Partnet: A large-scale benchmark for fine-grained and hierarchical part-level 3d object understanding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 909–918, 2019.

[83] Charlie Nash, Yaroslav Ganin, SM Ali Eslami, and Peter Battaglia. Polygen: An autoregressive generative model of 3d meshes. In *International conference on machine learning*, pages 7220–7229. PMLR, 2020.

[84] 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. *arXiv preprint arXiv:2003.06988*, 2020.

[85] Nelson Nauata, Sepidehsadat Hosseini, Kai-Hung Chang, Hang Chu, Chin-Yi Cheng, and Yasutaka Furukawa. House-gan++: Generative adversarial layout refinement network towards intelligent computational agent for professional architects. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13632–13641, 2021.

[86] Wamiq Para, Paul Guerrero, Tom Kelly, Leonidas Guibas, and Peter Wonka. Generative layout modeling using constraint graphs, 2020.

[87] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 165–174, 2019.

[88] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[89] Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron C. Courville. Film: Visual reasoning with a general conditioning layer. In *AAAI 2018*, 2018.

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

[91] Ben Poole, Ajay Jain, Jonathan T Barron, and Ben Mildenhall. Dreamfusion: Text-to-3d using 2d diffusion. *arXiv preprint arXiv:2209.14988*, 2022.

[92] Siyuan Qi, Yixin Zhu, Siyuan Huang, Chenfanfu Jiang, and Song-Chun Zhu. Human-centric indoor scene synthesis using stochastic grammar. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[93] Siyuan Qi, Yixin Zhu, Siyuan Huang, Chenfanfu Jiang, and Song-Chun Zhu. Human-centric indoor scene synthesis using stochastic grammar. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[94] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.

[95] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[96] Ali Razavi, Aaron Van den Oord, and Oriol Vinyals. Generating diverse high-fidelity images with vq-vae-2. *Advances in neural information processing systems*, 32, 2019.

[97] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *NIPS 2015*, 2015.

[98] Daniel Ritchie, Paul Guerrero, R Kenny Jones, Niloy J Mitra, Adriana Schulz, Karl DD Willis, and Jiajun Wu. Neurosymbolic models for computer graphics. *arXiv preprint arXiv:2304.10320*, 2023.

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

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

[101] RoomSketcher. Visualizing Homes. http://www.roomsketcher.com, 2017. Accessed: 2017-11-06.

[102] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[103] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[104] Manolis Savva, Angel X. Chang, Alexey Dosovitskiy, Thomas Funkhouser, and Vladlen Koltun. MINOS: Multimodal indoor simulator for navigation in complex environments. *arXiv:1712.03931*, 2017.

[105] Manolis Savva, Angel X. Chang, Pat Hanrahan, Matthew Fisher, and Matthias Nießner. SceneGrok: Inferring Action Maps in 3D Environments. In *SIGGRAPH Asia 2014*, 2014.

[106] Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, et al. Habitat: A platform for embodied ai research. *arXiv preprint arXiv:1904.01201*, 2019.

[107] Nadav Schor, Oren Katzir, Hao Zhang, and Daniel Cohen-Or. Componet: Learning to generate the unseen by part synthesis and composition. In *The IEEE International Conference on Computer Vision (ICCV)*, October 2019.

[108] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[109] Ari Seff, Yaniv Ovadia, Wenda Zhou, and Ryan P Adams. Sketchgraphs: A large-scale dataset for modeling relational geometry in computer-aided design. *arXiv preprint arXiv:2007.08506*, 2020.

[110] Pratheba Selvaraju, Mohamed Nabail, Marios Loizou, Maria Maslioukova, Melinos Averkiou, Andreas Andreou, Siddhartha Chaudhuri, and Evangelos Kalogerakis. Buildingnet: Learning to label 3d buildings. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10397–10407, 2021.

[111] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. CSGNet: Neural Shape Parser for Constructive Solid Geometry. *CoRR*, arXiv:1712.08290, 2017.

[112] Chao-Hui Shen, Hongbo Fu, Kang Chen, and Shi-Min Hu. Structure recovery by part assembly. *ACM Trans. Graph.*, 31(6), Nov. 2012.

[113] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. Learning structured output representation using deep conditional generative models. *Advances in neural information processing systems*, 28, 2015.

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

[115] Julian Straub, Thomas Whelan, Lingni Ma, Yufan Chen, Erik Wijmans, Simon Green, Jakob J. Engel, Raul Mur-Artal, Carl Ren, Shobhit Verma, Anton Clarkson, Mingfei Yan, Brian Budge, Yajie Yan, Xiaqing Pan, June Yon, Yuyang Zou, Kimberly Leon, Nigel Carter, Jesus Briales, Tyler Gillingham, Elias Mueggler, Luis Pesqueira, Manolis Savva, Dhruv Batra, Hauke M. Strasdat, Renzo De Nardi, Michael Goesele, Steven Lovegrove, and Richard Newcombe. The Replica dataset: A digital replica of indoor spaces. *arXiv preprint arXiv:1906.05797*, 2019.

[116] Yongbin Sun, Yue Wang, Ziwei Liu, Joshua Siegel, and Sanjay Sarma. Pointgrow: Autoregressively learned point cloud generation with self-attention. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 61–70, 2020.

[117] Minhyuk Sung, Hao Su, Vladimir G Kim, Siddhartha Chaudhuri, and Leonidas Guibas. ComplementMe: Weakly-supervised component suggestions for 3D modeling. *ACM Transactions on Graphics (TOG)*, 36(6):226, 2017.

[118] Benigno Uria, Marc-Alexandre Cote, Karol Gregor, Iain Murray, and Hugo Larochelle. Neural Autoregressive Distribution Estimation. *CoRR*, arXiv:1605.02226, 2016.

[119] Aaron Van Den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[120] Aaron van den Oord, Nal Kalchbrenner, Lasse Espeholt, Oriol Vinyals, Alex Graves, et al. Conditional image generation with pixelcnn decoders. In *Advances in Neural Information Processing Systems*, pages 4790–4798, 2016.

[121] Ashwin J. Vijayakumar, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *ICLR 2018*, 2018.

[122] Hao Wang, Nadav Schor, Ruizhen Hu, Haibin Huang, Daniel Cohen-Or, and Hui Huang. Global-to-local generative model for 3d shapes. *ACM Transactions on Graphics (Proc. SIGGRAPH ASIA)*, 37(6):214:1–214:10, 2018.

[123] Kai Wang, Paul Guerrero, Vladimir G Kim, Siddhartha Chaudhuri, Minhyuk Sung, and Daniel Ritchie. The shape part slot machine: Contact-based reasoning for generating 3d shapes from parts. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part III*, pages 610–626. Springer, 2022.

[124] 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 (TOG)*, 38(4):132, 2019.

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

[126] Kai Wang, Xianghao Xu, Leon Lei, Selena Ling, Natalie Lindsay, Angel X Chang, Manolis Savva, and Daniel Ritchie. Roominoes: Generating novel 3d floor plans from existing 3d rooms. In *Computer Graphics Forum*, volume 40, pages 57–69. Wiley Online Library, 2021.

[127] Yanzhen Wang, Kai Xu, Jun Li, Hao Zhang, Ariel Shamir, Ligang Liu, Zhiquan Cheng, and Yueshan Xiong. Symmetry hierarchy of man-made objects. In *Computer graphics forum*, volume 30, pages 287–296. Wiley Online Library, 2011.

[128] Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. DD-PPO: Learning near-perfect pointgoal navigators from 2.5 billion frames. *International Conference on Learning Representations (ICLR)*, 2020.

[129] Karl DD Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G Lambourne, Armando Solar-Lezama, and Wojciech Matusik. Fusion 360 gallery: A dataset and environment for programmatic cad construction from human design sequences. *ACM Transactions on Graphics (TOG)*, 40(4):1–24, 2021.

[130] Jiajun Wu, Chengkai Zhang, Tianfan Xue, Bill Freeman, and Josh Tenenbaum. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling. *Advances in neural information processing systems*, 29, 2016.

[131] Rundi Wu, Yixin Zhuang, Kai Xu, Hao Zhang, and Baoquan Chen. Pq-net: A generative part seq2seq network for 3d shapes. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[132] Wenming Wu, Lubin Fan, Ligang Liu, and Peter Wonka. Miqp-based layout design for building interiors. In *Computer Graphics Forum*, volume 37, pages 511–521. Wiley Online Library, 2018.

[133] 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 (TOG)*, 38(6):234, 2019.

[134] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1912–1920, 2015.

[135] Zhijie Wu, Xiang Wang, Di Lin, Dani Lischinski, Daniel Cohen-Or, and Hui Huang. Sagnet: Structure-aware generative network for 3d-shape modeling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2019)*, 38(4):91:1–91:14, 2019.

[136] Fei Xia, Amir R Zamir, Zhiyang He, Alexander Sax, Jitendra Malik, and Silvio Savarese. Gibson Env: Real-world perception for embodied agents. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9068–9079, 2018.

[137] Xiaohua Xie, Kai Xu, Niloy J. Mitra, Daniel Cohen-Or, Wenyong Gong, Qi Su, and Baoquan Chen. Sketch-to-design: Context-based part assembly. *Computer Graphics Forum*, xx(xx):xx, 2013.

[138] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *CoRR*, arXiv:1502.03044, 2015.

[139] Kun Xu, Kang Chen, Hongbo Fu, Wei-Lun Sun, and Shi-Min Hu. Sketch2scene: Sketch-based co-retrieval and co-placement of 3d models. In *SIGGRAPH 2013*, 2013.

[140] Kai Xu, Hao Zhang, Daniel Cohen-Or, and Baoquan Chen. Fit and diverse: Set evolution for inspiring 3d shape galleries. *ACM Transactions on Graphics, (Proc. of SIGGRAPH 2012)*, 31(4):57:1–57:10, 2012.

[141] Claudia Yan, Dipendra Kumar Misra, Andrew Bennett, Aaron Walsman, Yonatan Bisk, and Yoav Artzi. CHALET: cornell house agent learning environment. *CoRR*, arXiv:1801.07357, 2018.

[142] Ceyuan Yang, Yujun Shen, and Bolei Zhou. Semantic hierarchy emerges in deep generative representations for scene synthesis. *International Journal of Computer Vision*, 129:1451–1466, 2021.

[143] 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/CVF international conference on computer vision*, pages 4541–4550, 2019.

[144] Jie Yang, Kaichun Mo, Yu-Kun Lai, Leonidas J. Guibas, and Lin Gao. Dsm-net: Disentangled structured mesh net for controllable generation of fine geometry, 2020.

[145] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.

[146] Yi-Ting Yeh, Lingfeng Yang, Matthew Watson, Noah D. Goodman, and Pat Hanrahan. Synthesizing Open Worlds with Constraints Using Locally Annealed Reversible Jump MCMC. In *SIGGRAPH 2012*, 2012.

[147] Jiaxuan You, Bowen Liu, Rex Ying, Vijay S. Pande, and Jure Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. In *NeurIPS 2018*, 2018.

[148] Jiaxuan You, Rex Ying, Xiang Ren, William L. Hamilton, and Jure Leskovec. Graphrnn: A deep generative model for graphs. In *ICML 2018*, 2018.

[149] 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. In *SIGGRAPH 2011*, 2011.

[150] Richard Zhang, Phillip Isola, and Alexei A Efros. Colorful Image Colorization. In *ECCV 2016*, 2016.

[151] Yinda Zhang, Shuran Song, Ersin Yumer, Manolis Savva, Joon-Young Lee, Hailin Jin, and Thomas Funkhouser. Physically-based rendering for indoor scene understanding using convolutional neural networks. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[152] Zaiwei Zhang, Zhenpei Yang, Chongyang Ma, Linjie Luo, Alexander Huth, Etienne Vouga, and Qixing Huang. Deep generative modeling for scene synthesis via hybrid representations. *CoRR*, arXiv:1808.02084, 2018.

[153] Bolei Zhou, Agata Lapedriza, Aditya Khosla, Aude Oliva, and Antonio Torralba. Places: A 10 million image database for scene recognition. *IEEE transactions on pattern analysis and machine intelligence*, 40(6):1452–1464, 2017.

[154] Yang Zhou, Zachary While, and Evangelos Kalogerakis. Scenegraphnet: Neural message passing for 3d indoor scene augmentation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 7384–7392, 2019.

[155] C. Zou, E. Yumer, J. Yang, D. Ceylan, and D. Hoiem. 3D-PRNN: Generating Shape Primitives with Recurrent Neural Networks. In *ICCV 2017*, 2017.