Assembly of 3D Rooms into Floor Plans from Retrieved Layouts

LEON LEI
BROWN UNIVERSITY
MAY 14, 2020
ADVISOR: DANIEL RITCHIE

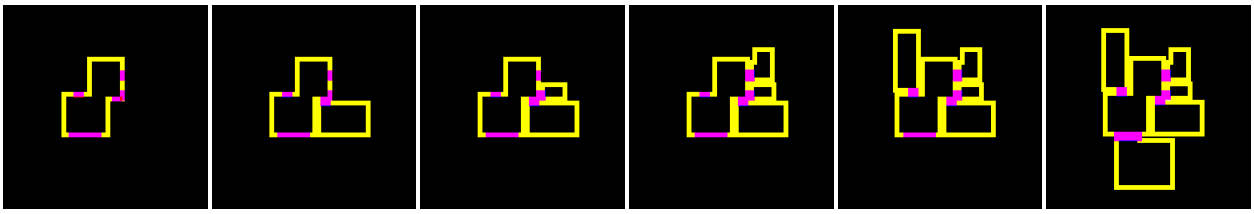

Fig. 1. Sequence of building up a floor plan. At each step, rooms are retrieved and their positions are optimized before insertion.

3D scene data is in high demand but current options are limited. Given this disparity, we propose to learn from existing floor plans and generate new data by retrieving and arranging rooms in novel ways, optimizing the geometry of the new layout, and editing existing 3D meshes to yield new 3D floor plans. This paper provides a high-level overview of the entire pipeline but focuses primarily on the design and implementation of the layout optimizer (Section 5) and its contribution to the overall pipeline.

1 INTRODUCTION

Despite an increasing demand for 3D scene data, current options are fairly limited. Synthetic data is usually not realistic enough, whereas scanned data is only available in limited quantities. The benefits of having large quantities of data are tremendous and can result in greatly improved results when training vision or reinforcement learning tasks. Despite the relative scarcity of 3D floor plan data, we noticed that 2D floor plan data is available in large quantities. Given this disparity, we propose a novel paradigm of retrieving, optimizing and editing rooms from existing layouts to assemble new structures.

2 BACKGROUND & RELATED WORK

Several previous works share the same goal of generating floor plans [Merrell et al. 2010] [Liu et al. 2013] [Wu et al. 2019], however our approach is intended to be more flexible and aims to assemble existing data rather than provide a method of unconstrained generation. In the context of the layout optimizer, our formulation draws most from [Wu et al. 2018]. Other projects in the realm of scene synthesis also share similar ideas to ours [Yu et al. 2011] [Fisher et al. 2012] [Li et al. 2018] [Zhang et al. 2018] [Wang et al. 2018] [Ritchie et al. 2019] [Wang et al. 2019], but floor plans constitute a different domain and thus require different methods.

3 OVERVIEW

The overall pipeline of our method can be seen in Figure 2. In this paper, we provide an overview of the entire pipeline but focus primarily on the contributions of the layout optimizer (Section 5).

In the first stage of the pipeline, we want to take existing 3D room data and use them to assemble novel configurations. Starting with a graph describing the floor plan, we retrieve rooms that are likely to match the graph along with a partially complete floor plan. This is done with a neural network that combines both graph and image-based features, and trained with a embedding loss similar to that in [Sung et al. 2017]. Since there is a limited amount of 3D room data, we first learn from a dataset consisting of 2D floor plans.

Next, since the layout retrieved by the previous stage of the pipeline is unlikely to have rooms that already fit together perfectly, we want to make changes to the room geometry by setting up and solving an optimization problem.

Finally, we can convert the generated floor plan to a full 3D scene by transferring the changes of the optimized 2D layout to its 3D counterpart. We propose two possible methods for doing this: either by applying a straightforward interpolation or by applying As-Rigid-As-Possible (ARAP) deformation to the original mesh.

4 ASSEMBLING FLOOR PLANS VIA ROOM RETRIEVAL

The overall idea is similar to [Wang et al. 2019], in which the relation graph is used as a guide and missing details are filled in. Directly finding rooms that fit together perfectly will leave each insertion with a very limited number of options and is also challenging to learn (would need to learn a joint parameterization of exact polygonal shapes and their locations). Instead, we adopt a metric learning perspective [Sung et al. 2017]. Instead of predicting rooms that will fit perfectly, we learn an embedding of rooms, such that similar rooms are grouped closer to each other. Given a partial floor plan, the goal is then to predict a distribution over the embedding space of rooms that will yield rooms that can fit well into the current floor plan. This is not going to be a perfect fit, so we need to run an additional optimization process after every step to resolve the conflicts.

In learning to retrieve rooms, we jointly train two networks, like [Sung et al. 2017]: The first of these is a retrieval network that takes the relation graph, as well as the current partial floor plan, and predicts a distribution over the likely rooms. The second of these is an embedding network that takes a room, and predicts its position

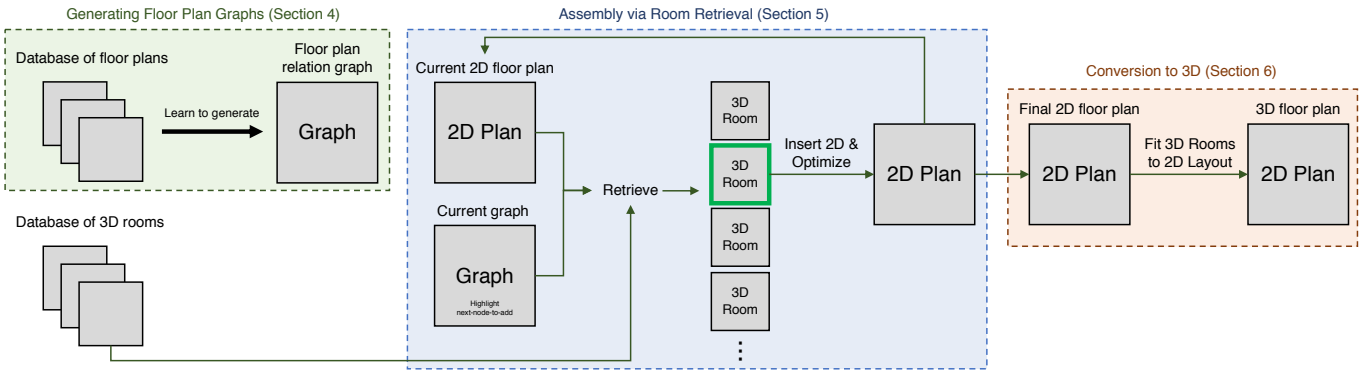

Fig. 2. Overall pipeline of our method. Floor plans are retrieved from the graph and inserted after an optimization step. The changes are then applied in 3D.

in the room embedding space. The two networks share the same embedding space, and are trained jointly.

5 OPTIMIZING RETRIEVED LAYOUTS

Given a list of rooms represented as 2D rectilinear polygons, the goal of the layout optimizer is to piece together these rooms in such a way so as to produce a valid final arrangement for the floor plan. Since the set of rooms selected by the retrieval network is likely to be a novel combination, it is unlikely that the selected rooms will naturally fit together as is. Furthermore, the graph representation obtained through the retrieval pipeline is not very sensitive to the geometry of the floor plan, and if left unchecked can result in a floor plan with various issues such as spatial gaps between rooms. To ensure that the final floor plan has certain intrinsic qualities such as non-overlap of rooms and adjacency of room portals, we solve an optimization problem to satisfy all imposed constraints while minimizing the overall distortion of rooms in the final layout. Our formulation requires mixed integer quadratic programming (MIQP) as a result of having a quadratic objective function and the need for Boolean variables in expressing the constraints.

5.1 Basic Formulation

We can start with a simplifying assumption and only consider rooms that are rectangles rather than arbitrary 2D rectilinear polygons. In our solver, we express each rectangle i in terms of 4 variables: its upper-left vertex position $\langle x_i, y_i \rangle$, its width w_i, and its height h_i.

Objective function: We want the initial arrangement of rooms to distort as little as possible. There are two options for this: we could either choose to just penalize changes in the width/height of rooms, or we could also choose to penalize changes to the positions of rooms in addition to their widths/heights. Depending on the usage and desired results, both approaches are reasonable, and we set up our objective function to minimize the amount of distortion in both instances. The amount of distortion can be defined as a least squares, i.e. the sum of the squared differences between the initial and final values of variables. Thus, our objective is simply:

$$\text{minimize } \|A\mathbf{x} - \mathbf{b}\|_2^2 \tag{1}$$

where A is the identity matrix, $\mathbf{x}$ is a vector of variables, and $\mathbf{b}$ is a corresponding vector of initial values.

We also define the following categories of constraints similar to those in [Wu et al. 2018], for all rectangles $1 \leq i \leq N$ where N is the number of rectangles:

Non-negativity: We define all variables to be non-negative so that our solution lies in the positive quadrant and no output rectangles have a negative width and height:

$$x_i, y_i, w_i, h_i \geq 0 \tag{2}$$

Bounding box: We want our solution to lie within a provided bounding box, i.e. no corners of any rectangles fall outside of the predefined range. Let x_{max}, y_{max} represent the upper-bounds of the range. Then, the constraints can be expressed as:

$$\begin{cases} x_i \leq x_{max} \\ y_i \leq y_{max} \\ x_i + w_i \leq x_{max} \\ y_i + h_i \leq y_{max} \end{cases} \tag{3}$$

Non-overlap: We require the solution to have no overlapping rectangles between any given pair of rooms 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. For example, $D = L$ means that i is to the left of j. Expressed as constraints, these are:

$$\begin{cases} x_i - w_j \geq x_j, \text{when D = R} \\ x_i + w_i \leq x_j, \text{when D = L} \\ y_i - h_j \geq y_j, \text{when D = B} \\ y_i + d_i \leq y_j, \text{when D = T} \end{cases} \tag{4}$$

However, we can't apply these constraints all at once because at the time of optimization we won't know which of the 4 relationships would result in the best arrangement. Instead, we let the optimizer select for this relationship by introducing an auxiliary binary variable $\sigma_{i,j}^D$. When $\sigma_{i,j}^D = 1$, rectangles i and j have the relationship D.

This results in the following set of constraints for each pair i, j:

$$\begin{cases} x_i - w_j \geq x_j - M \cdot (1 - \sigma^R_{i,j}) \\ x_i + w_i \leq x_j + M \cdot (1 - \sigma^L_{i,j}) \\ y_i - h_j \geq y_j - M \cdot (1 - \sigma^B_{i,j}) \\ y_i + d_i \leq y_j + M \cdot (1 - \sigma^T_{i,j}) \\ \sum_{D=1}^{4} \sigma^D_{i,j} \geq 1 \end{cases} \tag{5}$$

where M is a large constant to ensure that rectangles i and j do not overlap in direction D when $\sigma^D_{i,j} = 1$. When $\sigma^D_{i,j} = 0$, the inequality is always satisfied, so we need to add the last constraint which requires that at least one of the four auxiliary variables has a value of 1. M is set to $x_{max} + y_{max}$ in our implementation.

Rectangle adjacency: Since the output of the retrieval pipeline includes adjacency relationships between rooms, e.g. the living room is adjacent to the bedroom, we want the optimizer to be able to enforce these adjacencies in the final solution. To achieve this, we add constraints so that the two rooms will overlap, and in combination with the non-overlap constraint this forces the overlap to occur only on the edges of the rooms. Additionally, it can be useful to specify the length of overlap $L_{i,j}$ between two rooms for instance if there is a door or opening along the walls of the room. As with the non-overlap constraints, we can introduce a binary variable $\theta_{i,j}$ to determine the direction of connection, where $\theta_{i,j} = 1$ means the connection is vertical and otherwise it is horizontal.

$$\begin{cases} x_i \leq x_j + w_j - L_{i,j} \cdot \theta_{i,j} \\ x_i + w_i \geq x_j + L_{i,j} \cdot \theta_{i,j} \\ y_i \leq y_j + h_j - L_{i,j} \cdot (1 - \theta_{i,j}) \\ y_i + h_i \geq y_j + L_{i,j} \cdot (1 - \theta_{i,j}) \end{cases} \tag{6}$$

Ignore rooms: We also provide the ability to specify optional constraints that fix the size and position of a room in the final solution. This is useful from the perspective of building a room layout iteratively, where we have already determined an optimal arrangement for some number of rooms and are now trying to add an additional room to the existing layout. These variables could also be refactored out of the optimization problem, although the effect is negligible. An experiment was conducted which removed these variables and constraints from the optimization problem, with no noticeable difference in solve times. Given initial values $\{x'_i, y'_i, w'_i, h'_i\}$, the constraints are simply to set each variable of an ignored room to be equal to its initial value.

Next, we expand our set constraints to handle rooms represented as arbitrary 2D rectilinear polygons and not just as rectangles.

5.2 Maximal Decomposition of Rooms into Rectangles

To simplify and make our constraints easier to express, our algorithm first takes the initial set of rooms represented as arbitrary 2D rectilinear polygons and computes a maximal rectangle decomposition (See Figure 3). A maximal rectangle decomposition has the property such that two adjacent rectangles will always share their edge completely. This makes it easy to impose additional constraints to bind the decomposed rectangles of a given room together in the final solution. In combination with the individual constraints on each rectangle, the final solution preserves the properties of the room while also satisfying all the other constraints imposed on rectangles. Without this initial decomposition step, constraints such as non-overlap quickly become unfeasible to express linearly, because unlike in rectangles, the shapes of arbitrary rooms are non-convex.

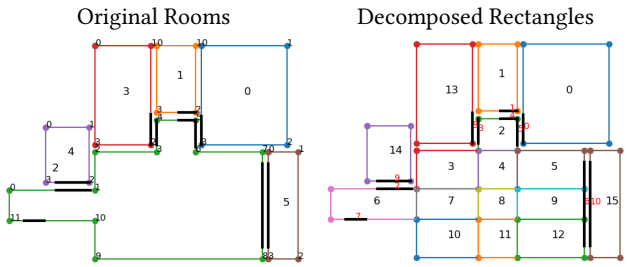

Fig. 3. Example of a floor plan before and after decomposition

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

$$\begin{cases} x_i + w_i = x_j \\ y_i = y_j \\ h_i = h_j \end{cases} \tag{7}$$

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):

$$\begin{cases} y_i + h_i = y_j \\ x_i = x_j \\ w_i = w_j \end{cases} \tag{8}$$

Algorithm for computing maximal decomposition: Given a list of vertices representing a room, a simple algorithm to produce a maximal rectangle decomposition is as follows: Construct a grid by building a set of all x-positions and a set of all y-positions from the vertices of the input list, and iterate over each rectangle-cell of the grid. Run a check to determine whether this rectangle-cell lies inside the polygon defined by the input room, and add it to the resulting set if so. While this is done, a list of the vertical and horizontal adjacencies between rectangles in the grid can also be saved for the purposes of expressing decomposition constraints.

After the optimization problem is solved, the resulting room can then be reconstructed by undoing the decomposition of rectangles.

5.3 Room level constraints

In addition to the constraints provided for rectangles, it is useful to have higher-level controls on the scale of rooms. Up until now, we have been able to express adjacencies between two rectangles i, j, but the same notion of adjacency becomes more difficult to express when the rooms are not necessarily rectangles. One possibility is to explicitly enforce constraints on the decomposed rectangles within

the rooms, but this is not always practical from a caller perspective. For instance, if room A consists of three decomposed rectangles, which of these should a rectangular room B be made adjacent to? The optimal solution could involve an adjacency to one, two, or even all three rectangles. Furthermore, room A could also be composed of multiple rectangles, and we may want any number of those to be adjacent. The goal is for the caller to be able to just specify that rooms A and B are adjacent, and have the optimizer select for the best possible adjacency.

Room adjacency: Expanding on the non-overlap and adjacency constraints defined for rectangles, we define the following set of constraints for each i, j combination of decomposed rectangles such that $i \in$ room A and $j \in$ room B to handle the possibility of any given pair of decomposed rectangles being adjacent to one another. The constraints are then expressed as follows:

$$\begin{cases} x_i \le x_j + w_j - L_{i,j} \cdot \theta_{i,j} + M \cdot (1 - \sigma_{i,j}) \\ x_i + w_i \ge x_j + L_{i,j} \cdot \theta_{i,j} - M \cdot (1 - \sigma_{i,j}) \\ y_i \le y_j + h_j - L_{i,j} \cdot (1 - \theta_{i,j}) + M \cdot (1 - \sigma_{i,j}) \\ y_i + h_i \ge y_j + L_{i,j} \cdot (1 - \theta_{i,j}) - M \cdot (1 - \sigma_{i,j}) \end{cases} \tag{9}$$

and then for each pair of adjacent rooms A, B, we add the following constraint over all valid combinations i, j of rectangles:

$$\sum \sigma_{i,j} \ge 1 \tag{10}$$

where $\theta_{i,j}$ is a binary variable that determines the direction of connection (horizontal or vertical), $\sigma_{i,j}$ is a binary variable that determines whether or not there is a connection between rooms i, j, and $L_{i,j}$ is the minimum length of overlap between i, j. As with the non-overlap constraints, we require in the last constraint that the adjacency of at least one combination of decomposed rectangles i, j within rooms A and B is non-trivially satisfied.

Modified Objective: A feature of most floor plans is that rooms are not merely adjacent to each other but usually fit together as snugly as possible so as not to introduce voids in the floorplan. With this in mind, we can use a modified objective function that attempts to maximize the total value of $\sum \sigma_{i,j}$ across all room adjacencies in the floor plan. The intuition is that we want the binary variable $\sigma_{i,j}$ to have a value of 1 as frequently as possible. Let Σ be the number of $\sigma_{i,j}$ that are assigned a value of 1 in the optimization. Our modified objective can then be expressed as:

$$\text{minimize } \|A\mathbf{x} - \mathbf{b}\|_2^2 - \alpha\Sigma \tag{11}$$

where α is a constant that weights the relative importance of the term in the objective function. In our implementation, we set $\alpha = 50$.

5.4 Expressing Portal information

In addition to fitting together the shapes of the rooms, the layout optimizer is also expected to produce floor plans that satisfy several portal constraints. We define a portal as any hole/opening in the wall(s) of a room, although for our purposes we are most concerned with portals in the context of doorways or passageways that connect to other rooms. This is crucial because we want to produce not just individual rooms but a complete floor plan (i.e. one that a navigation agent could walk through). Therefore, we need to be able to specify in our layout which rooms' portals line up and have the optimizer produce a final result that reflects these constraints. In our model, we describe a portal i as a line with centroid position $\langle px_i, py_i \rangle$ and radius pr_i that can slide along any of the walls in a room, and add three additional variables to our optimization problem for each portal in the floor plan. Then we can introduce the following constraints for portals:

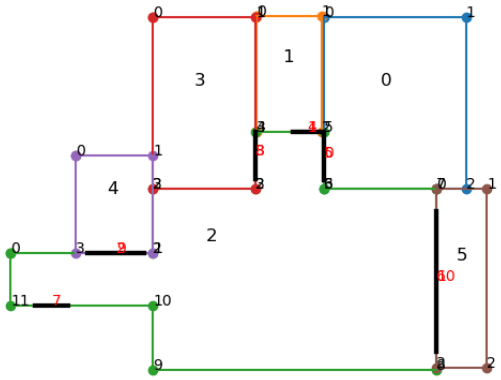

Fig. 4. Optimized output of Figure 3 with adjacency constraints applied

Portal sliding: We require that portals stay on the same wall that they are initially defined to be on, and that their new position/length do not extend beyond the range of the wall that they lie on. We first handle the case where a room is just a rectangle. Since we know which wall D the portal lies on ahead of time (for example, $D = T$ means the portal lies on the top wall), we can express the constraints in four cases as follows:

$$\begin{cases} \text{if D = T} \begin{cases} py_i = y_i \\ px_i \ge x_i + pr_i \\ px_i \le x_i + w_i - pr_i \end{cases} \\ \text{if D = B} \begin{cases} py_i = y_i + h_i \\ px_i \ge x_i + pr_i \\ px_i \le x_i + w_i - pr_i \end{cases} \\ \text{if D = L} \begin{cases} px_i = x_i \\ py_i \ge y_i + pr_i \\ py_i \le y_i + h_i - pr_i \end{cases} \\ \text{if D = R} \begin{cases} px_i = x_i + w_i \\ py_i \ge y_i + pr_i \\ py_i \le y_i + h_i - pr_i \end{cases} \end{cases} \tag{12}$$

However, since our optimization operates on variables corresponding to decomposed rectangles rather than entire rooms, we need to be able to handle the case where a portal may lie on a room wall that is shared by more than one decomposed rectangle. This essentially

uses the same constraints as defined above, but requires us to know the indices of the rooms between which the portal can slide. For instance, if a portal 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 rectangle, then the constraints for this case would be as follows:

$$\begin{cases} px_i = x_i \\ py_i \geq y_i + pr_i \\ py_i \leq y_k + h_k - pr_i \end{cases} \tag{13}$$

In our implementation, we determine the indices of these rectangles at the same time as when we execute the decomposition step, and dynamically build up these constraints from those indices.

Portal adjacency: If two portals are defined to be adjacent to each other, then we want their centroid positions and radii to be equivalent. This constraint can be expressed simply as follows for each pair i, j of adjacent portals:

$$\begin{cases} px_i = px_j \\ py_i = py_j \\ pr_i = pr_j \end{cases} \tag{14}$$

Objective function: Since we want the optimized portals to change as little as possible from their initial values in both position and length, we include all new variables representing portals in our least squares objective.

5.5 Implementation Details

We implemented our algorithm in Python using the CVXPY interface and used Gurobi to solve the MIQP. We tested several solvers that are able to solve mixed-integer programs and found that Gurobi produced the most favorable results for our problem formulation in both speed and optimality.

Using the CVXPY interface: CVXPY made it easy to express the constraints of our problem in a natural way. In converting our problem from paper to code, we found it most effective to set up a separate array for each category of constraints and dynamically build them up before concatenating and sending the final array as input to the solver. In representing our variables using CVX containers, we separated the position variables from the length variables for ease of indexing and so that we could easily switch between using both types of variables or just the variables corresponding to lengths in our objective function. For our objective function, we were able to take advantage of CVXPY's `sum_squares` method to express our problem as a least squares.

6 CONVERTING FLOOR PLANS TO 3D SCENES

After the layout optimizer has produced a final layout for the floor plan, the final step is to take a room corresponding to the original layout and edit it to match the specifications of the optimizer. Namely, the positions and lengths of walls and portals may have changed between the original and final layout, so we want to deform the geometry of the original mesh such that it respects the new 2D outline while simultaneously minimizing the non-rigid distortion applied to semantically meaningful objects. Although we can use any 3D scene data, the results we present currently use the Matterport3D dataset.

Next, we present high-level descriptions of two possible approaches that can be used to achieve this conversion: one that involves a straightforward interpolation and another that uses ARAP deformation.

Straightforward interpolation: Along with the output from the optimizer, we can use bilinear interpolation to determine the new positions of vertices in the mesh. This is facilitated by the fact that the optimizer already computes a rectangle decomposition for all rooms, which aids in the interpolation step. For all vertices with a label that corresponds to a non-rigid object, e.g. *wall*, *floor*, or *ceiling*, we can just move them to their new interpolated positions. For all other vertices corresponding to rigid objects, e.g. *pillow*, *bed*, *tv*, we can just compute how much the centroid position of this object has moved via interpolation and translate the entire object by the change in centroid positions. Further enhancements to this approach include the cutting out of objects from the mesh before moving them in a separate step to avoid stretching effects, as well as the grouping of objects in close proximity as one entity to avoid unnecessary distancing (e.g. a *bed* and a *bedside* table). Figure 5 shows a sample of preliminary results that includes the method of cutting out objects but not their grouping (e.g. pillow and bed are still treated as separate entities).

ARAP deformation: Another possible approach that is less naive than the straightforward interpolation method is to use As-Rigid-As-Possible (ARAP) techniques to deform the mesh. First, a preprocessing step selects the largest connected component in the mesh as required by ARAP. Control points are then determined and selected along the walls of the room and moved to their interpolated new positions. Rigid objects are treated as such and their vertices are just moved to their new positions in a similar manner to the straightforward interpolation approach. ARAP then solves for the positions of the remaining vertices in the mesh keeping the overall mesh as rigid as possible. One benefit of the ARAP approach is that it does not introduce holes as a result of cutting objects out of the mesh. More importantly, it is guaranteed to never cause inversions in the mesh, whereas the straightforward interpolation approach provides no such guarantees. For instance, in an extreme case where a wall compresses around a window, the straightforward approach would simply move the wall to overlap the window, whereas ARAP would squeeze and wrap the mesh of the wall around the window.

7 RESULTS

The overall pipeline is capable of producing a large variety of visually reasonable results. More specifically, the layout optimizer described in Section 5 works effectively and produces reasonable results across a wide range of inputs. Within the context of the overall pipeline, the optimization step for inserting a single room into the floor plan takes around 2-3 minutes, although there are instances of problems with difficult solutions that can cause the optimizer to exhaust its allotted time limit. These instances are not considered for final floor

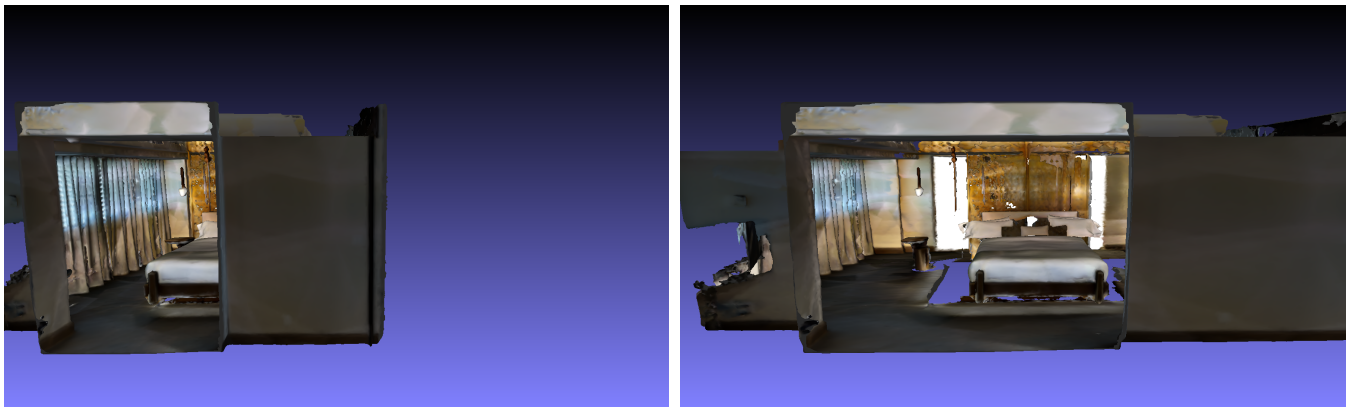

Fig. 5. Preliminary results of a room mesh edited with bilinear interpolation that has been intentionally moved and stretched to an extreme degree.

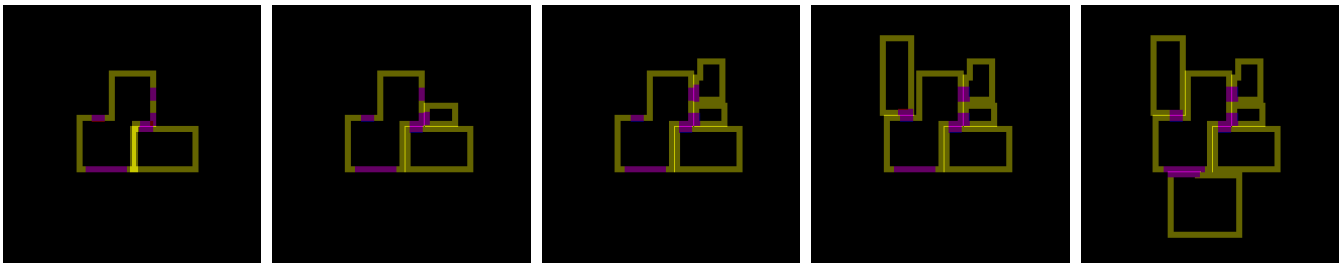

Fig. 6. Pre-optimized results corresponding to Figure 1 selected by the retrieval network. Some overlaps, gaps, and varying portal lengths are apparent.

plans as they likely correspond to infeasible arrangements in the first place.

Figure 6 shows the generated sequences of a sample floor plan presented in Figure 1, taken from before the layout optimizer is applied. The optimizer is able to successfully fix all the obvious issues in these pre-optimized floor plans, including the removal of gaps and the lining up of portals at each step.

8 CONCLUSION

The main contribution of this work is a novel paradigm of retrieving, optimizing, and editing existing rooms to create new 3D floor plan data. In the context of the layout optimizer, the formulation of the room arrangement problem as a MIQP allowed us to generate valid floor plans that satisfied the various qualities we imposed with the least amount of deformation to the original rooms. These optimized results can then be used for the editing and stitching of 3D room meshes into floor plans.

One of the main limitations of our method is its inability to handle complex room geometries. Although the bulk of available data can be characterized by rectilinear shapes, future work in handling other room geometries would make our method more robust. From the perspective of the layout optimizer, in addition to the possibility of handling more complex room geometries, additional work can be done looking into speeding up its runtime. Although analyzing the runtime of MIQP programs is a challenging task, a faster optimizer would aid in the process of generating large amounts of floor plan data in a shorter time. One consideration may be to implement the project in another language such as C++, since the Gurobi-CVXPY interface has been reported to be slower. Another consideration is to modify the formulation of the problem. For example, when decomposing rooms into rectangles, it may be possible to compute a *minimal nonoverlapping cover* (MNC) instead of a maximal decomposition. A MNC is a partition with the fewest possible number of rectangles, and this would reduce the number of rectangles in the optimization problem, hopefully resulting in faster solve times. However, the constraints under such a method would become significantly more complicated to express, and it's possible that would conversely contribute to slower solve times. Additionally, algorithms for computing a MNC are also more complex than that of a maximal decomposition, and bring with them their own runtime considerations.

Lastly, a good deal of future work for this project can be done in refining the mesh editing approaches in Section 6. This would entail producing more realistic floor plans that have convincing object arrangements and are seamlessly connected with other rooms in the layout of the floor plan. With ample new data in hand, we also open up the possibility for future experiments, e.g. the use of our model for training navigation agents.

REFERENCES

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

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

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

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

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

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

Kai Wang, Yu-An Lin, Ben Weissmann, Manolis Savva, Angel X Chang, and Daniel Ritchie. 2019. PlanIt: Planning and instantiating indoor scenes with relation graph and spatial prior networks. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 132.

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

Wenming Wu, Lubin Fan, Ligang Liu, and Peter Wonka. 2018. MIQP-based Layout Design for Building Interiors. In *Computer Graphics Forum*, Vol. 37. Wiley Online Library, 511–521.

Wenming Wu, Xiao-Ming Fu, Rui Tang, Yuhan Wang, Yu-Hao Qi, and Ligang Liu. 2019. Data-driven interior plan generation for residential buildings. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 234.

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

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