Supplementary Materials for Infinigen Indoors: Photorealistic Indoor Scenes using Procedural Generation

Alexander Raistrick*, Lingjie Mei*, Karhan Kayan*, (*equal contribution; random order)
David Yan, Yiming Zuo, Beining Han, Hongyu Wen, Meenal Parakh,
Stamatis Alexandropoulos, Lahav Lipson, Zeyu Ma, Jia Deng
Department of Computer Science, Princeton University

Contents

1. Constraint Specification API

1.1. Design Goals & Related Work

Although Infinigen Indoors provides default settings to generate data in one click, we also intend for it to serve as a general-purpose framework for creating customized indoor datasets of any variety. This extends to creating indoor environments besides just residential homes, such as public or commercial spaces. This also includes creating data for *user-specific* versions of these spaces according to known requirements, for example, creating warehouses or grocery stores with structures that follow the design guidelines of some specific company. We also expect that users might have similarly detailed requirements about smaller objects in a scene, like the organization of ornaments on a shelf, or appliances in a commercial kitchen. These needs motivate several attributes in which our solver system differs from and improves over prior work:

No pre-arranged scenes expected. Our system assumes that most users do *not* already have a collection of pre-arranged 3D scenes (e.g. annotated 3D scans or artist-arranged scenes) matching their specific use case.

This is in contrast to prior arrangement solvers [5, 7, 20], which tune many low-level distributions against pre-arranged 3D scene datasets such as [5, 7]. To extend such a system to a new domain, the intended workflow would be to manually arrange or to scan-and-annotate many indoor scenes, then extract regularities to interpolate into new room arrangements. Of course, one could manually specify coefficients in any system, but systems designed around pre-arranged scenes invariably have lower-level, less-interpretable constraints, such as full probability distributions or dense pairwise potentials. Automatically tuning coefficients is possible under our framework, but is not required, and we expect fewer scenes due to the reduced number of free variables.

This assumption - no pre-arranged input data - also maintains Infinigen Indoors's guarantee of 100% customizability and that no *hidden* biases are possible, since everything is derived from open-source code. A system tuned against a fixed dataset would inherit any bias or unexpected properties from that dataset.

High-level constraints. To make this primarily user-driven workflow feasible, our constraint language provides a higher-level (more abstract; less verbose) API than prior systems. As explained in Sec. 1.2, we allow rules to be specified

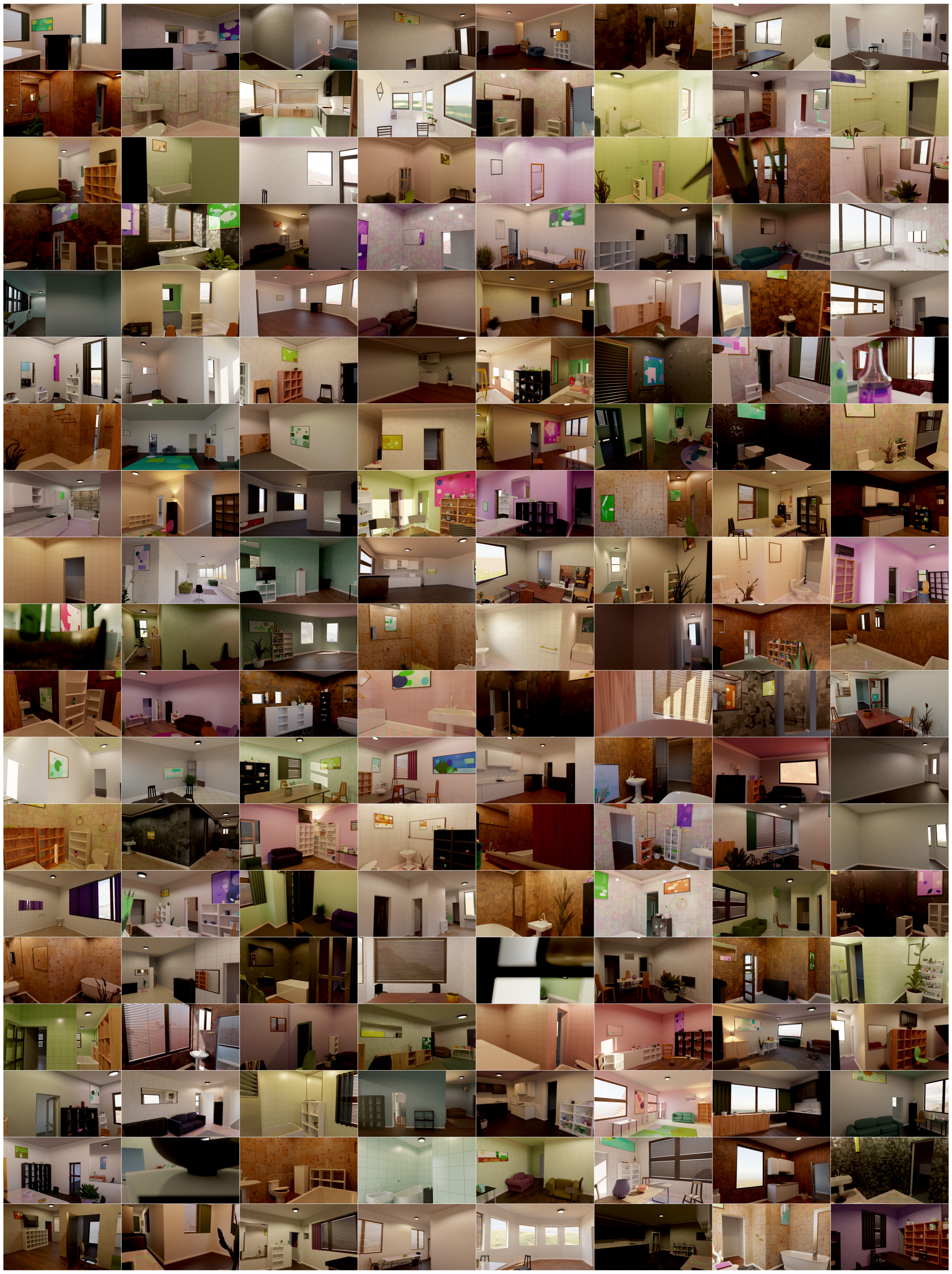


Figure 1. Random, non-cherry-picked sample of procedurally generated residential homes (Part 1 of 2)

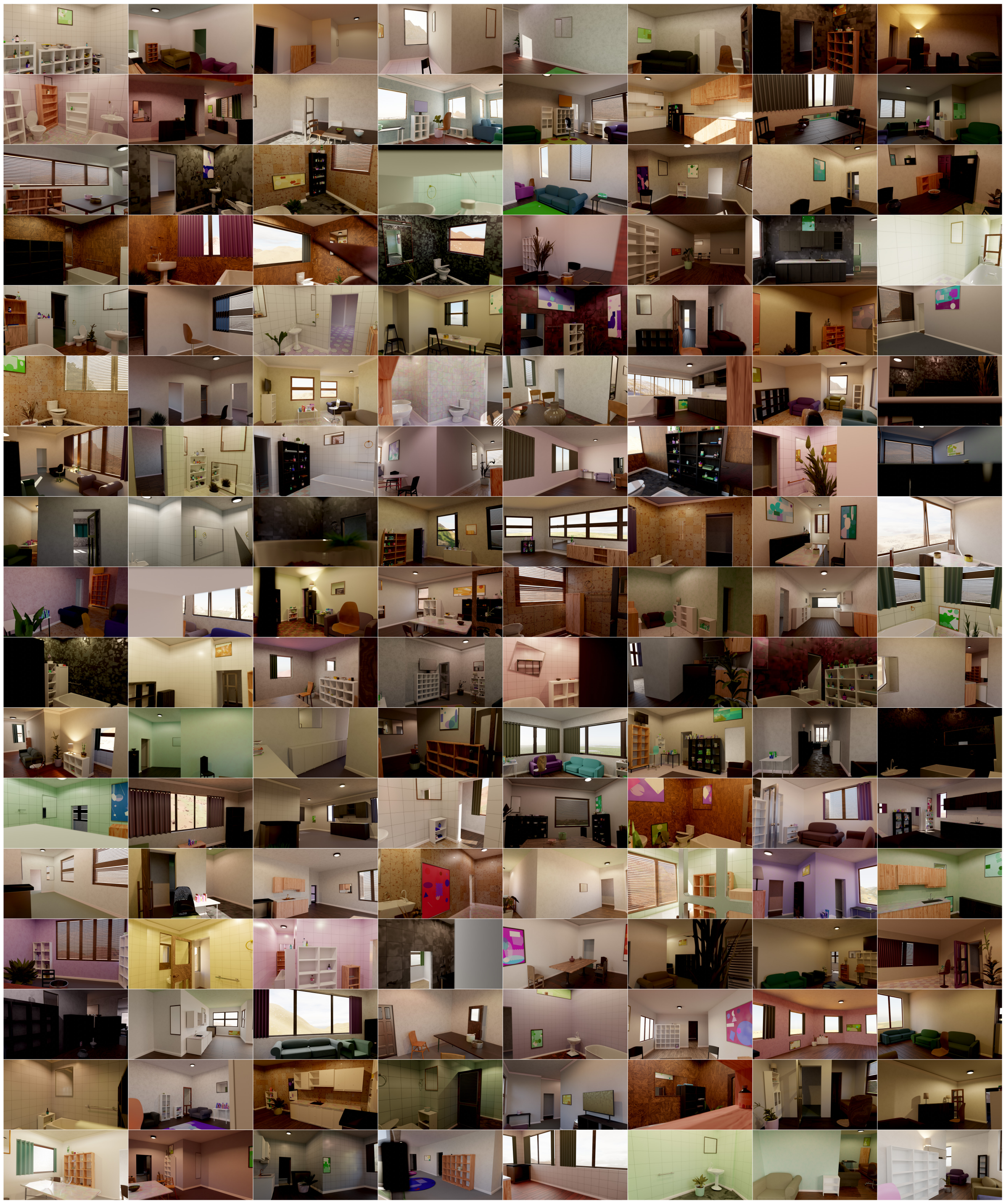
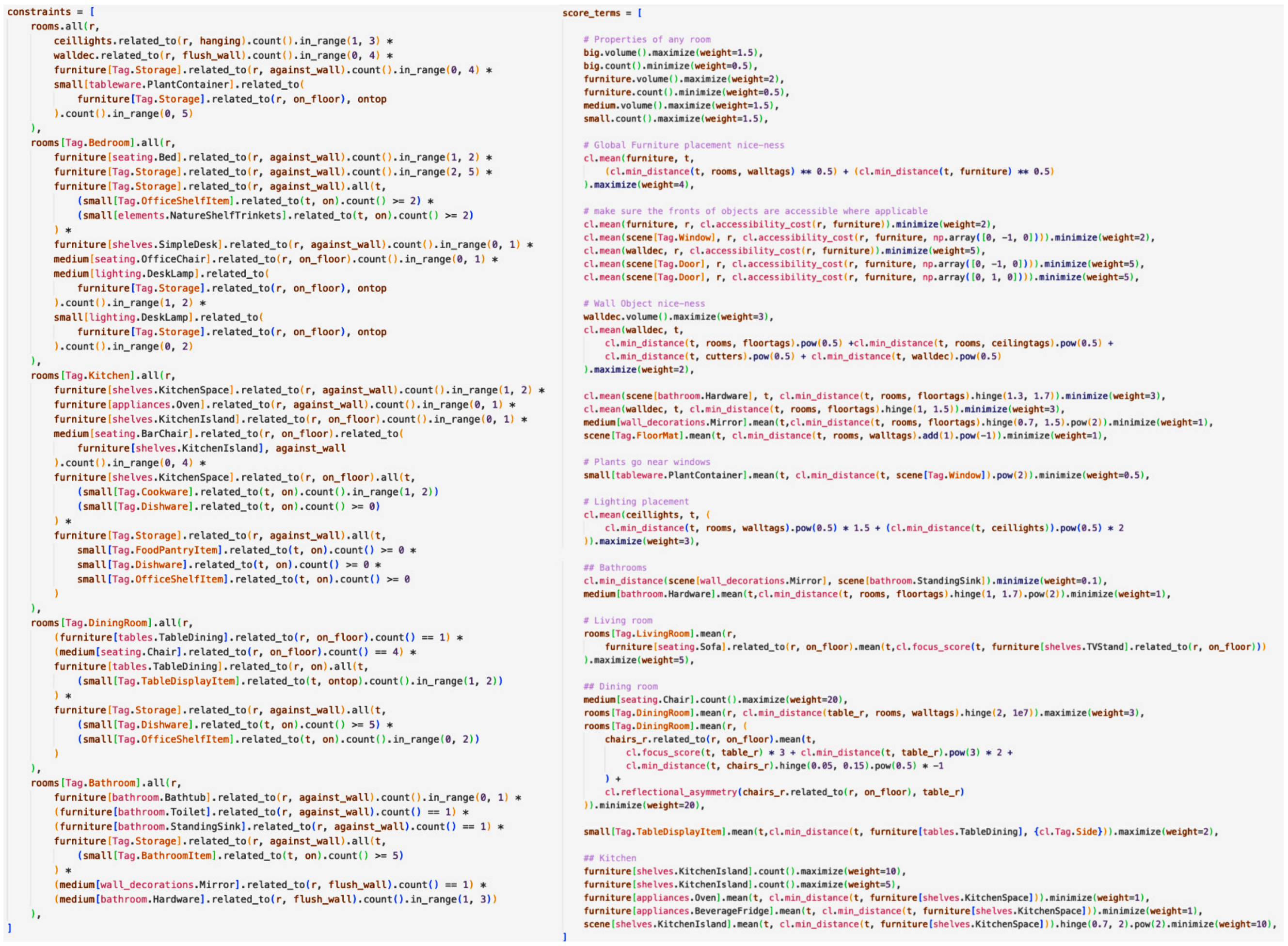


Figure 2. Random, non-cherry-picked sample of procedurally generated residential homes (Part 2 of 2)



```
constraints = [
    rooms.all(r,
        ceillights.related_to(r, hanging).count().in_range(1, 3) *
        walldec.related_to(r, flush_wall).count().in_range(0, 4) *
        furniture[Tag.Storage].related_to(r, against_wall).count().in_range(0, 4) *
        small[tableware.PlantContainer].related_to(
            furniture[Tag.Storage].related_to(r, on_floor), ontop
        ).count().in_range(0, 5)
    ),
    rooms[Tag.Bedroom].all(r,
        furniture[seating.Bed].related_to(r, against_wall).count().in_range(1, 2) *
        furniture[Tag.Storage].related_to(r, against_wall).count().in_range(2, 5) *
        furniture[Tag.Storage].related_to(r, against_wall).all(t,
            (small[Tag.OfficeShelfItem].related_to(t, on).count() >= 2) *
            (small[elements.NatureShelfTrinkets].related_to(t, on).count() >= 2)
        ) *
        furniture[shelves.SimpleDesk].related_to(r, against_wall).count().in_range(0, 1) *
        medium[seating.OfficeChair].related_to(r, on_floor).count().in_range(0, 1) *
        medium[lighting.DeskLamp].related_to(
            furniture[Tag.Storage].related_to(r, on_floor), ontop
        ).count().in_range(1, 2) *
        small[lighting.DeskLamp].related_to(
            furniture[Tag.Storage].related_to(r, on_floor), ontop
        ).count().in_range(0, 2)
    ),
    rooms[Tag.Kitchen].all(r,
        furniture[shelves.KitchenSpace].related_to(r, against_wall).count().in_range(1, 2) *
        furniture[appliances.Oven].related_to(r, against_wall).count().in_range(0, 1) *
        furniture[shelves.KitchenIsland].related_to(r, on_floor).count().in_range(0, 1) *
        medium[seating.BarChair].related_to(r, on_floor).related_to(
            furniture[shelves.KitchenIsland], against_wall
        ).count().in_range(0, 4) *
        furniture[shelves.KitchenSpace].related_to(r, on_floor).all(t,
            (small[Tag.Cookware].related_to(t, on).count().in_range(1, 2))
            (small[Tag.Dishware].related_to(t, on).count() >= 0)
        ) *
        furniture[Tag.Storage].related_to(r, against_wall).all(t,
            small[Tag.FoodPantryItem].related_to(t, on).count() >= 0 *
            small[Tag.Dishware].related_to(t, on).count() >= 0 *
            small[Tag.OfficeShelfItem].related_to(t, on).count() >= 0
        )
    ),
    rooms[Tag.DiningRoom].all(r,
        (furniture[tables.TableDining].related_to(r, on_floor).count() == 1) *
        (medium[seating.Chair].related_to(r, on_floor).count() == 4) *
        furniture[tables.TableDining].related_to(r, on).all(t,
            (small[Tag.TableDisplayItem].related_to(t, ontop).count().in_range(1, 2))
        ) *
        furniture[Tag.Storage].related_to(r, against_wall).all(t,
            (small[Tag.Dishware].related_to(t, on).count() >= 5) *
            (small[Tag.OfficeShelfItem].related_to(t, on).count().in_range(0, 2))
        )
    ),
    rooms[Tag.Bathroom].all(r,
        furniture[bathroom.Bathtub].related_to(r, against_wall).count().in_range(0, 1) *
        (furniture[bathroom.Toilet].related_to(r, against_wall).count() == 1) *
        (furniture[bathroom.StandingSink].related_to(r, against_wall).count() == 1) *
        furniture[Tag.Storage].related_to(r, against_wall).all(t,
            (small[Tag.BathroomItem].related_to(t, on).count() >= 5)
        ) *
        (medium[wall_decorations.Mirror].related_to(r, flush_wall).count() == 1) *
        (medium[bathroom.Hardware].related_to(r, flush_wall).count().in_range(1, 3))
    ),
]
```

```
score_terms = [

    # Properties of any room
    big.volume().maximize(weight=1.5),
    big.count().minimize(weight=0.5),
    furniture.volume().maximize(weight=2),
    furniture.count().minimize(weight=0.5),
    medium.volume().maximize(weight=1.5),
    small.count().maximize(weight=1.5),

    # Global Furniture placement nice-ness
    cl.mean(furniture, t,
        (cl.min_distance(t, rooms, walltags) ** 0.5) + (cl.min_distance(t, furniture) ** 0.5)
    ).maximize(weight=4),

    # make sure the fronts of objects are accessible where applicable
    cl.mean(furniture, r, cl.accessibility_cost(r, furniture)).minimize(weight=2),
    cl.mean(scene[Tag.Window], r, cl.accessibility_cost(r, furniture, np.array([0, -1, 0]))).minimize(weight=2),
    cl.mean(walldec, r, cl.accessibility_cost(r, furniture)).minimize(weight=5),
    cl.mean(scene[Tag.Door], r, cl.accessibility_cost(r, furniture, np.array([0, -1, 0]))).minimize(weight=5),
    cl.mean(scene[Tag.Door], r, cl.accessibility_cost(r, furniture, np.array([0, 1, 0]))).minimize(weight=5),

    # Wall Object nice-ness
    walldec.volume().maximize(weight=3),
    cl.mean(walldec, t,
        cl.min_distance(t, rooms, floortags).pow(0.5) +cl.min_distance(t, rooms, ceilingtags).pow(0.5) +
        cl.min_distance(t, cutters).pow(0.5) + cl.min_distance(t, walldec).pow(0.5)
    ).maximize(weight=2),

    cl.mean(scene[bathroom.Hardware], t, cl.min_distance(t, rooms, floortags).hinge(1.3, 1.7)).minimize(weight=3),
    cl.mean(walldec, t, cl.min_distance(t, rooms, floortags).hinge(1, 1.5)).minimize(weight=3),
    medium[wall_decorations.Mirror].mean(t,cl.min_distance(t, rooms, floortags).hinge(0.7, 1.5).pow(2)).minimize(weight=1),
    scene[Tag.FloorMat].mean(t, cl.min_distance(t, rooms, walltags).add(1).pow(-1)).minimize(weight=1),

    # Plants go near windows
    small[tableware.PlantContainer].mean(t, cl.min_distance(t, scene[Tag.Window]).pow(2)).minimize(weight=0.5),

    # Lighting placement
    cl.mean(ceillights, t, (
        cl.min_distance(t, rooms, walltags).pow(0.5) * 1.5 + (cl.min_distance(t, ceillights)).pow(0.5) * 2
    )).maximize(weight=3),

    ## Bathrooms
    cl.min_distance(scene[wall_decorations.Mirror], scene[bathroom.StandingSink]).minimize(weight=0.1),
    medium[bathroom.Hardware].mean(t,cl.min_distance(t, rooms, floortags).hinge(1, 1.7).pow(2)).minimize(weight=1),

    # Living room
    rooms[Tag.LivingRoom].mean(r,
        furniture[seating.Sofa].related_to(r, on_floor).mean(t,cl.focus_score(t, furniture[shelves.TVStand].related_to(r, on_floor)))
    ).maximize(weight=5),

    ## Dining room
    medium[seating.Chair].count().maximize(weight=20),
    rooms[Tag.DiningRoom].mean(r, cl.min_distance(table_r, rooms, walltags).hinge(2, 1e7)).maximize(weight=3),
    rooms[Tag.DiningRoom].mean(r, (
        chairs_r.related_to(r, on_floor).mean(t,
            cl.focus_score(t, table_r) * 3 + cl.min_distance(t, table_r).pow(3) * 2 +
            cl.min_distance(t, chairs_r).hinge(0.05, 0.15).pow(0.5) * -1
        ) +
        cl.reflectional_asymmetry(chairs_r.related_to(r, on_floor), table_r)
    )).minimize(weight=20),

    small[Tag.TableDisplayItem].mean(t,cl.min_distance(t, furniture[tables.TableDining], {cl.Tag.Side})).maximize(weight=2),

    ## Kitchen
    furniture[shelves.KitchenIsland].count().maximize(weight=10),
    furniture[shelves.KitchenIsland].count().maximize(weight=5),
    furniture[appliances.Oven].mean(t, cl.min_distance(t, furniture[shelves.KitchenSpace])).minimize(weight=1),
    furniture[appliances.BeverageFridge].mean(t, cl.min_distance(t, furniture[shelves.KitchenSpace])).minimize(weight=1),
    scene[shelves.KitchenIsland].mean(t, cl.min_distance(t, furniture[shelves.KitchenSpace])).hinge(0.7, 2).pow(2).minimize(weight=10),
]
```

Figure 3. Constraint Program for whole residential homes. Left shows hard constraints, right shows continous objective scores. Our system flexibly composes API calls to apply any constraint to any class of object describable within the scene.

over abstract classes of objects (”furniture”, ”seating”, ”diningchair”), rather than for specific meshes or furniture categories. Whereas other systems directly specify distributions over geometric quantities (e.g. the distance/angle to wall distribution between every unique furniture type [20]), we allow these to distributions to arise from competing ergonomic objectives (dining tables are generally as far away from walls as possible, but are forced closer in more cramped apartments due to competition from other furniture)

General-purpose, extensible constraint language. We aim for our system to cover all possible indoor arrangement problems, with the whole vision community contributing useful constraint programs or solver improvements as high-level python code.

Despite this, our current set of operations likely does not cover all possible use cases, so we have designed our system to be extensible with arbitrary additional operations. All existing operations are implemented as simple Numpy, Trimesh or Blender-API functions, and any control flow or hard-constraints operations can optionally be easily added to our constraint-graph reasoning utilities as explained in Sec. 5.2.

1.2. API Description

`with_semantics` (also notated as operator square-brackets for brevity) extracts the subset of a set of objects that satisfies a semantic predicate, e.g. ”extract the subset of rooms which are dining-rooms” or ”extract the subset of furniture objects which are shelving”. The hierarchy of these predicates is defined by asset creators, or can be reconfigured by constraint program writers if they intend to use an object for an unusual purpose. Using hierarchical classes for this filtering operation allows every constraint to apply to the most broad class of objects possible, to avoid rewriting or restating constraints for objects that fulfill a similar function.

Optimization objectives based on...	Our API Function	Example Usage (Described in Natural Language)
Objects with (abstract) semantics	`with_semantics` a.k.a. [ ]	Scope a constraint to a hierarchical class e.g. shelves, storage, all furniture, or all objects
Objects related to other objects	`related_to`	Cooking pots go in the center when on tables, but can go anywhere on a countertop.
Objects on arbitrary surfaces	`SupportedBy`	Multi-story homes, decorations on shelves, countertops, fridges
Whether objects overhang	`StableAgainst`	Glassware should be secure, paintings cant overhang walls
Non-convex object shapes	Yes, procedural placeholders	Objects go inside shelves, chairs tuck under table
Variable quantity of objects	`count`	Allow between 0 and 3 sofas in a living-room, but as many as possible
Size of objects	`area`, `volume`	Generate the biggest possible TV & Sofa that fits well
Pair-wise distances	`min_distance`	Place dining tables & ceiling lights far from walls
Pair-wise angle difference	`angle_alignment`	Align tables to parallel to the nearest wall
Symmetry around an object	`rotational_asymmetry`	Chairs should be rotationally symmetric when placed around circular tables
Symmetry across a plane	`reflection_asymmetry`	Bed-side tables should be symmetric on either side of a bed
Objects facing other objects	`focus_score`	Sofas should face TVs or paintings
Object accessibility	`accessibility_cost`	Leave space in front of sofas / appliances
Empty space on a surface	`freespace_2d`	Leave some space leftover in room / on countertop
Arbitrary arithmetic / nonlinearities	`+ - * / pow hinge`	Encourage certain ratio of ceiling-lights to room-area
Boolean comparisons / logic	`== < <= and in_range`	Ensure there are 2 to 6 chairs for every table
every object must satisfy a predicate	`all`	Every bookcase must have $\geq$ 10 books
sum/mean across specific objects	`mean sum`	Compute average distance to wall over many objects, rather than minimum

Table 1. Capabilities covered by our API. Please see Sec. 2 and 3 for example programs, and surrounding text for full descriptions. For our API, functions can be composed arbitrarily, e.g. `scene.with_semantics(...).related_to(...).count().hinge(...)` to create a nonlinear objective w.r.t. number of objects in a certain context. For other works, we cite any related work that fulfills each capability in any capacity.

`related_to` extracts the subset of a set of objects related to any member of a second set of objects via some relation. The exact way in which the objects are related is user-configurable by passing in any parameterized `Relation` object from the options below (StableAgainst, SupportedBy), which represents a predicate that can be True or False of any pair of objects.

By combining `related_to` with other filtering operations, the user can express constraints on arbitrarily complex contexts such as "maximize the number of dining chairs against dining tables inside of rooms adjacent to kitchens", to encourage plenty of seating near food preparation areas, etc.

`scene` retrieves the set of all objects currently in the scene. All constraint programs are ultimately functions of the current scene state, so this node serves as the leaf node of all constraint program expressions (besides numeric constants). Users rarely place constraints on `scene` directly. Instead, we expect the user to first take subsets via the abovementioned operations.

`StableAgainst` specifies a relation using a child object's planar surface, a parent object's planar surface, and a margin between the surfaces. It checks that the child's surface is parallel to the parent's, the child is not overhanging, and the child's surface is exactly at the specified margin. A concrete example would be specifying the sofa as stable against the floor with zero margin and stable against the wall with a 10cm margin. Alternatively, specifying a painting to be stable against the wall ensures that the painting does not overhang across the edge of the wall.

`SupportedBy` specifies a relation using a child object's planar surface and a parent object's planar surface. It means that the child object would not fall over from the parent object. More precisely, the surfaces are parallel against each other with zero margins, and the centroid of the child object is contained within the convex hull of the intersection between the child and the parent object. The last condition is to ensure zero torque by gravity. An example use case is a coffee cup teetering on the table's edge. In this case, the cup is supported by the table, but it is not stable against it since it is overhanging.

`count` returns the cardinality of a set of objects in the scene.

`area, volume` returns the total area or volume of the bounding boxes of objects in a set. We use bounding boxes to avoid expensive calculations to compute the exact volume of each mesh, and we find this serves as a suitable proxy to incentivize larger assets. Area always takes over the two largest axes of an object and is usually used for 2D objects like paintings or rugs.

`min_distance` calculates the minimum distance between two sets of tagged objects. For instance, the minimum distance between the walls and the back of the couch. The minimum distance is defined as the distance between the closest two points on the two sub-meshes identified by the tags.

`angle_alignment_cost` quantifies how far a group of objects are from being angle aligned to a reference object on the XY plane. The cost is calculated as

$$\sum_i \frac{1-\cos\theta_i}{2}$$

where θ_i is the angular difference between the front-facing normal of object i and the inward normal of the closest edge of the reference object. The contribution of each object is in the $[0, 1]$ range.

An example use case is minimizing the angle alignment cost between chairs and tables to make the chairs face the table. Another example is using an alignment score to align furniture to the walls in order to give the arrangement a more grid-like shape.

`rotation_asymmetry` gives a continuous characterization of the rotational asymmetry of a set of objects based on [6, 21]. It measures the deviation of the set of objects from a regular polygon with perfectly rotationally symmetric orientations. From another perspective, it measures the rotational asymmetry of a set of point-vector pairs. The score consists of two parts and is calculated as

$$\text{score} = \frac{\text{location asymmetry} + \text{orientation asymmetry}}{2}.$$

Suppose the location of the ith object is given by $\vec{x}_i$ and there are n objects. The location asymmetry is calculated as follows:

1. Let $\vec{p}_i = \vec{x}_i - c$ where c is the centroid of the objects.
2. Rotate all $\vec{p}_i$ so that $\vec{p}_1$ is aligned with the axis.
3. Normalize $\vec{p}_i$ by dividing by $\max_i ||\vec{p}_i||$.
4. Let $\vec{f}_i$ be vector $\vec{p}_i$ rotated by $-2i\pi/n$.
5. Compute $\vec{q}$ as the average of $\vec{f}_i$.
6. Let $\vec{w}_i$ be vector $\vec{q}$ rotated by $2i\pi/n$.
7. Then, we have location asymmetry $= \frac{1}{n}\sum ||\vec{w}_i - \vec{p}_i||^2$.

The orientation asymmetry score follows the same steps as the location asymmetry, but with $\vec{p}_i$ replaced by the frontal plane normal of the object i.

As an example, rotational asymmetry score can be used to encourage tableware being rotationally symmetric on the table not only with respect to their location but also their orientation. It can also be used to make chairs rotationally symmetric around the table.

`reflection_asymmetry` calculates a continuous reflectional asymmetry score for a set of objects relative to a reference object. This score quantifies the deviation of objects from mirror symmetry. The process involves reflecting each object across a plane and then comparing the original and reflected objects. From another perspective, it quantifies the mirror asymmetry of a set of point-vector pairs. The asymmetry score is computed as follows:

1. **Determine Reflection Plane:** Identify the plane of reflection, which can be any of the median planes of the bounding box of the reference object.

2. **Reflect Objects:** The objects O_i are represented by $(\vec{p}_i, q_i)$ where $\vec{p}_i$ is the object's location and q_i is the object's orientation. Each object O_i is reflected across the plane to obtain its mirror image O'_i. The reflection of a point $\vec{p}$ is given by $\vec{p'} = \vec{p} - 2(\vec{p}\cdot\vec{N}_{plane})\vec{N}_{plane}$. The reflection of an axis-angle represented orientation $\theta\vec{e}$ is given by $\theta'\vec{e'}$ where $\vec{e'} = \vec{e} - 2(\vec{e}\cdot\vec{N}_{plane})\vec{N}_{plane}$ and $\theta' = -\theta$.

3. **Bipartite Matching:** A cost-minimizing bipartite matching is performed between the set of original objects $\{O_i\}$ and their reflected counterparts $\{O'_i\}$ to find the optimal pairings based on a cost matrix derived from positional and angular deviations. We use a modified Jonker-Volgenant algorithm for this step [2, 10].

4. **Calculate Deviations:**
   - **Positional Deviation:** For each paired object $(O_i, O'_i) = ((\vec{p}_i, q_i), (\vec{p'}_i, q'_i))$, calculate the Euclidean distance $D_{pos} = ||\vec{p}_i - \vec{p'}_i||$.
   - **Angular Deviation:** Calculate the angular difference $D_{ang} = 2\arccos(|q_i \cdot q'_i|)$, where q_i and q'_i are the quaternion representations of the paired objects' orientations.

5. **Weight Deviations:** Each deviation is weighted by a factor $V(O_i)$, which is the volume of the bounding box of O_i. The weighted deviation for each object pair is $D_{dev}(O_i) = V(O_i) \times (D_{pos} + D_{ang})$.

6. **Normalization:** The total deviation is normalized by a factor α, which is the average distance between objects: $\alpha = \frac{1}{N(N-1)}\sum_{i\neq j} ||\vec{p}_i - \vec{p'}_i||$, where N is the number of objects.

7. **Compute Asymmetry Score:** The reflectional asymmetry score is derived as

$$\text{Score} = 1 - \frac{1}{1 + \sum_i D_{dev}(O_i)/\alpha}$$

This reflection score is useful in contexts such as encouraging chairs to be symmetric around a long rectangular table, or encouraging furniture to have mirror symmetry for visual appeal, or encouraging paintings to be symmetrical across the room.

`accessibility_cost` computes how much a set of objects B block access to a set of objects A. We offer two versions. In the fast version, the function selects the closest object in B to each object in A based on the centroid distance. In the slow version, it finds the closest point on any mesh in B to each mesh in A. The mathematical formulation can be described as follows:
We first take the projection of a's centroid onto its specified plane (frontal plane by default) by

$$\vec{a}_{\text{proj}} = \vec{a}_c - \left((\vec{a}_c - \vec{f}_p) \cdot \vec{n}_a\right) \vec{n}_a$$

where $\vec{a}_c$ is the centroid of object a, $\vec{f}_p$ is a point on the specified plane, and $\vec{n}_a$ is the normal vector of the specified plane.
For a given object $a \in A$, we define $\vec{b}(a)$ and $\vec{b}_{\text{closest\_pt}}$. The fast version defines

$$\vec{b}(a) = \arg\min_{b \in B} ||\vec{b}_c - \vec{a}_{\text{proj}}||$$
$$\vec{b}_{\text{closest\_pt}} = \text{Centroid of the selected } \vec{b}(a)$$

The slow version defines

$$\vec{b}(a) = \text{Object in } B \text{ with the point closest to mesh } a$$
$$\vec{b}_{\text{closest\_pt}} = \text{Point on mesh } \vec{b}(a) \text{ closest to mesh } a$$

For both fast and slow versions, the accessibility cost is calculated as

$$\text{cost} = \sum_{a \in A} \frac{(\vec{b}_{\text{closest\_pt}} - \vec{a}_{\text{proj}}) \cdot \vec{n}_a}{||\vec{b}_{\text{closest\_pt}} - \vec{a}_{\text{proj}}||^2} \times ||\vec{b}(a)_d||$$

where $\vec{b}(a)_d$ is the diagonal vector of the bounding box of the chosen object $\vec{b}(a)$. We note that the accessibility cost increases as the blocking object gets larger, as the blocking objects get closer, and as the blocking object is more in front of the specified plane.

An example usage of accessibility cost is when we want to penalize objects being directly in front of TVs, paintings, or closets.

`focus_score` encourages focusing a set of objects A on an object b. It is calculated as

$$\sum_{a \in A} \frac{1 - \vec{n}_a \cdot (\vec{b}_c - \vec{a}_c)}{2||\vec{b}_c - \vec{a}_c||}$$

where $\vec{n}_a$ is the front facing normal of object a. The vectors $\vec{a}_c, \vec{b}_c$ denote the centroids of a and b respectively. The contribution of each object is in the $[0, 1]$ range.

An example use case of focus score is focusing the sofas on the TV to encourage a more realistic layout. Another example use case is focusing a set of seats on a round table.

`freespace_2d` returns the amount of 2D free space available on a set of objects A after accounting for the space occupied by objects B. It is calculated as

$$\sum_{a \in A} \mu(\text{proj}(a)) - \sum_{b \in B} \mu(\text{proj}(b))$$

where proj is the projection to the XY plane and μ gives the area of a 2D shape. An example use case is minimizing the free space on a table to encourage placing more objects on the table, or maximizing the free space in a living room to make it less cluttered.

Arithmetic / non-linearities provide basic scalar arithmetic and an implementation of the standard hinge loss function, all computed using the standard Python definitions. The exact set of mathematical operators provided here is not critical; our system treats scalar losses as a black box, so any arbitrary Python math expressions are acceptable.

Boolean comparisons allow equality or inequality checking between values, usually for creating hard constraints on cardinalities or distances. When used to check the size of a set, our system will use the constraint statement to inform what Addition moves are proposed as explained in "Cardinality Bounding".

`all` provides control flow logic akin to the "forall" $\forall$ symbol as used in formal proofs. It is commonly used in constructing soft/hard constraints. We avoid allowing arbitrary Python control flow (for loops, if statements, etc.) as it makes symbolic reasoning difficult by restricting the user to symbolic expressions. This design decision is similar to other compute graph programming frameworks (e.g. Tensorflow [1], CVXPY [4]). For example, by forcing the user to use a symbolic 'all' statement rather than a for loop, we can make inferences such as "if all chairs go near tables, and there must be at least one chair, then there must be at least one table", which allows the user to write higher level and fewer constraints, with the system deducing all logical consequences.

Forall statements take as input a loop variable name, and a constraint program that contains the loop variable as a leaf node at one or more locations. During execution, the child constraint program is substituted with the real values of the loop variable and evaluated to obtain the various results.

`mean, sum` compute the standard scalar mean and sum operations, using similar control flow logic and evaluation substitution mechanisms as `all` as described above.

2. Extended Random Sample & Constraint Code for Residential Scenes

Please inspect Figures 1 and 2 for an extended random sample of our main residential home generator (as seen previously in Figure 1 of the main paper).

These images were derived from the constraint code designed for residential homes, shown in Figure 3. We have a total number of 105 soft and hard constraints, with 19 for dining rooms, 14 for living rooms, 9 for bathrooms, 18 for kitchens, 16 for warehouses, and 30 designed for general purposes. These constraints are used to cover object assignments (object A goes on object B), ratios (numbers of chairs per table, objects per shelf), stability (TV placed against wall; objects don't overhang unsupported), distance (plants placed near window), and more. This initial set creates scenes that are both visually compelling and useful for training. It is easy to add more scene types, as little as 15 lines of code each.

3. Extension to warehouse scenes

To show the generality of our solving system, we implemented a simple constraint program that uses existing language features to specify the high-level objectives of a warehouse environment, with furniture on shelves and smaller items on wooden pallets. See Fig. 4 for the full program. Various further extensions are possible, for example indicating a preference for larger objects to be placed lower or higher on the shelves, or certain objects to be placed near the front of the warehouse / store. We show example images in Fig. 5, as well as a topdown view showing only the shelving and lighting layout.

4. Floorplan Solver Details

4.1. Floor plan graph generation

We generate floor plans that have 1 to 3 floors. For each floor, we generate a floor plan graph where individual nodes represent a room with a certain type, and each edge represents the connectedness of two rooms it is linked to. We support rooms of the following type: *kitchen*, *bedroom*, *living-room*, *closet*, *hallway*, *bathroom*, *garage*, *balcony*, *dining-room*, *utility*, *staircase*. *hallway* can mean any corridor or passage between rooms, and *staircase* means the room or space where one can find the stairs.

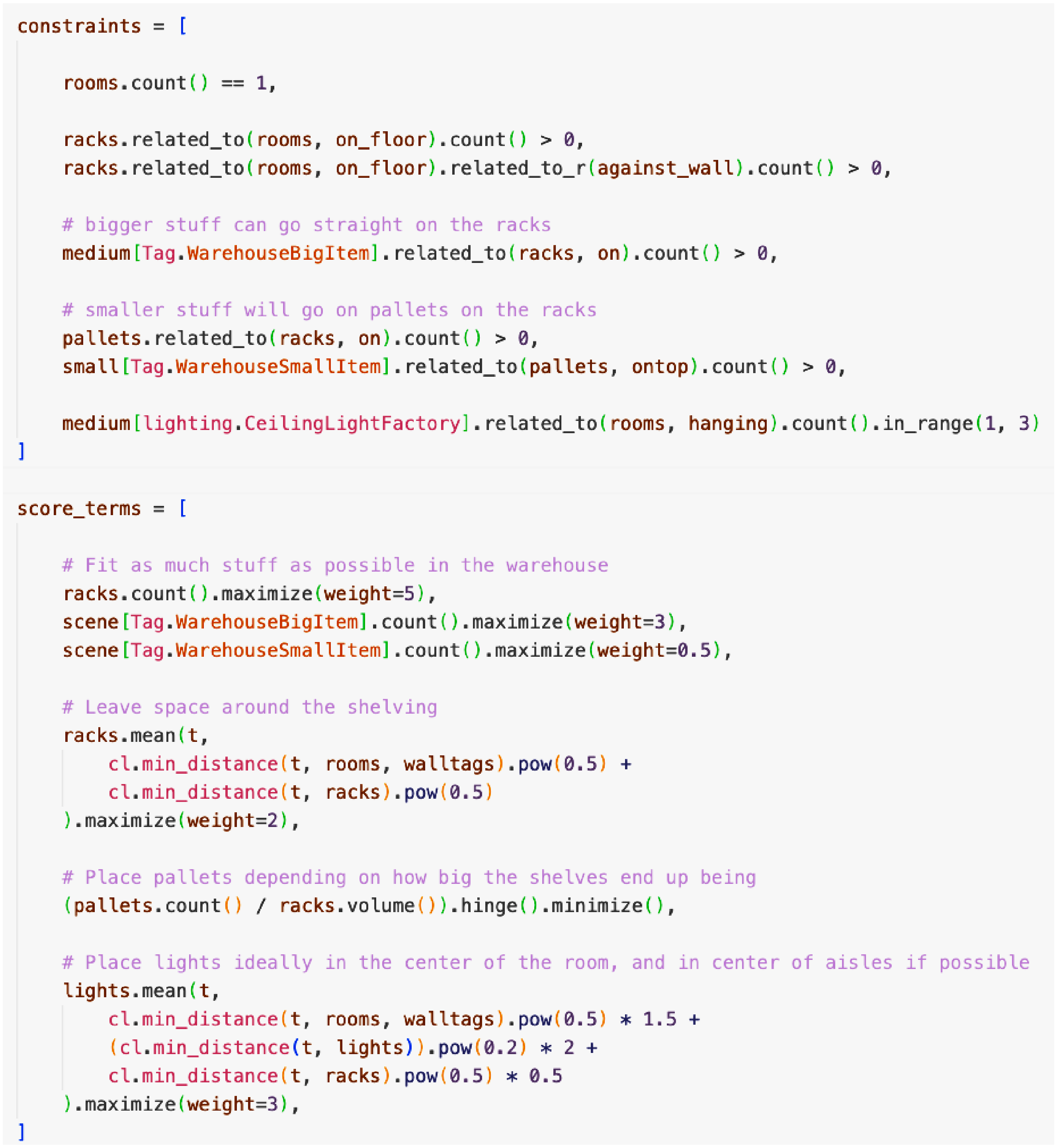
The graph is generated by a Probabilistic Context-free Grammar, where the graph first starts off as a single node *living-room*, and gradually appends zero, one or more rooms of certain types to the leaf nodes. The probability distribution that we use is shown in Tab. 2 and Tab. 3. Additional Edges are added to rooms to create a floor plan graph based on the generated tree. Additional hallways are added and shared among the children with the same parent. Based on the number of floors and the current level, a porch (*balcony*) or *staircase* may also be added to the graph. All room plans that do not observe bathroom privacy (i.e. a bedroom is connected to a bathroom without going through other bedrooms) or are not planar are rejected. Based on the user input, floor plans with an incorrect number of designated rooms are also rejected. By default, we require all floor plan graphs to have at least one living room and one bathroom.

Room parent	Room children	Probability
LivingRoom	LivingRoom	0.1
	Bedroom	$\text{Cat}(0, 0.3, 0.3, 0.3, 1)$
	Closet	0.1
	Bathroom	0.4
	Garage	0.4
	Balcony	0.2
	DiningRoom	0.8
	Utility	0.2
	Hallway	$\text{Cat}(0.5, 0.4, 0.1)$
Kitchen	Garage	0.1
	Utility	0.2
Bedroom	Bathroom	0.3
	Closet	0.5
Bathroom	Closet	0.2
DiningRoom	Kitchen	1.0
	Hallway	0.2

Table 2. Probability of the numbers of rooms PCFG produces for each leaf node in the graph for the ground floor. Such probability is conditioned on the parent room type(Column 1) and the children room type(Column 2). The probability(Column 3) can either be a Bernoulli distribution (shown as the sole parameter) or a Categorical(Cat) distribution (shown as the probability on the number of children, starting with zero).

Room parent	Room children	Probability
LivingRoom	Bedroom	$\text{Cat}(0, 0.4, 0.5, 0.2)$
	Closet	0.2
	Bathroom	0.4
	Balcony	0.4
	Utility	0.2
	Hallway	$\text{Cat}(0, 0.5, 0.5)$
Bedroom	Bathroom	0.3
	Closet	0.5
Bathroom	Closet	0.2
Balcony	Utility	0.4
	Hallway	0.1

Table 3. Probability of the numbers of rooms PCFG produces for each leaf node in the graph. The annotations are similar to Tab. 2

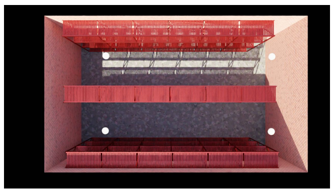


```
constraints = [

    rooms.count() == 1,

    racks.related_to(rooms, on_floor).count() > 0,
    racks.related_to(rooms, on_floor).related_to_r(against_wall).count() > 0,

    # bigger stuff can go straight on the racks
    medium[Tag.WarehouseBigItem].related_to(racks, on).count() > 0,

    # smaller stuff will go on pallets on the racks
    pallets.related_to(racks, on).count() > 0,
    small[Tag.WarehouseSmallItem].related_to(pallets, ontop).count() > 0,

    medium[lighting.CeilingLightFactory].related_to(rooms, hanging).count().in_range(1, 3)
]

score_terms = [

    # Fit as much stuff as possible in the warehouse
    racks.count().maximize(weight=5),
    scene[Tag.WarehouseBigItem].count().maximize(weight=3),
    scene[Tag.WarehouseSmallItem].count().maximize(weight=0.5),

    # Leave space around the shelving
    racks.mean(t,
        cl.min_distance(t, rooms, walltags).pow(0.5) +
        cl.min_distance(t, racks).pow(0.5)
    ).maximize(weight=2),

    # Place pallets depending on how big the shelves end up being
    (pallets.count() / racks.volume()).hinge().minimize(),

    # Place lights ideally in the center of the room, and in center of aisles if possible
    lights.mean(t,
        cl.min_distance(t, rooms, walltags).pow(0.5) * 1.5 +
        (cl.min_distance(t, lights)).pow(0.2) * 2 +
        cl.min_distance(t, racks).pow(0.5) * 0.5
    ).maximize(weight=3),
]
```

Figure 4. Constraint program for warehouse scenes. In only a few high-level statements, we specify the heirarchy of allowed objects, and competing placement objectives that give rise to an appropriate shelf and object layout for any warehouse scene.

4.2. Floor plan initialization

Based on the floor plan graph for a specific floor, we first deduct an estimated contour area based on the sum of typical areas of all the rooms on one floor, which can also used to derive the width and length of the contour. To derive the contour on one floor, we randomly bevel the corners with a rectangular, round, or 45-degree profile that provides the


(a) Shelving Arrangement

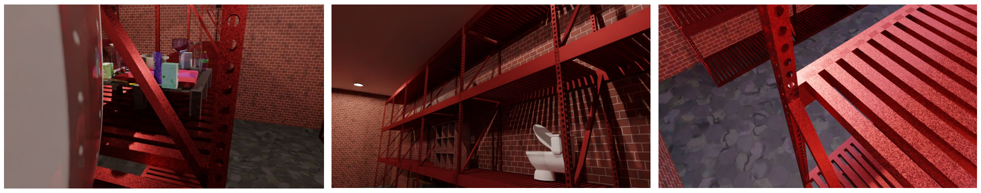

(b) Example Images

Figure 5. Warehouse scene arrangement (left) and example first-person images (right). Using only a few high-level objectives, we extend our existing placement system and existing furniture generators to create a hardware-store-like environment.

diversity of the contour shape. Contours for floors upstairs are either the exact same copy of the contour on its lower floor or a subset of the contour on its lower floor.

The spaces are subdivided from the contour following the Mondrian Process [18]. For each iteration, we randomly select a mostly rectangular space and divide it along one of its axes, and we repeat such division so that there are 1.5 times more blocks than are required in the floor plan graph. All divisions apply by rounding off the division onto a grid with size 0.5, and divisions leading to a bad aspect ratio are rejected. We merge the spaces until there's the same number of spaces as in the floor plan graph, then compute the adjacency relations of all divided spaces, where spaces are adjacent if they share an edge of size greater or equal to a threshold (to place doors). We randomly add a staircase placeholder inside the contour for multistory floor plans, which roughly indicates the location of the staircase. The staircase placeholder ensures staircases across adjacent floors are in the same spatial location.

Among these contour divisions, we try to find one where the assignment of rooms suffices the floor plan graph via adjacency relations. In addition to the adjacency relations in the floor plan graph, we also ensure that all exterior-facing rooms, including the bedroom, garage, and balcony, have access to the house's exterior. Only the divided spaces intersecting with the staircase placeholder can be assigned to the staircase room. We can find a proper assignment of rooms that satisfies the floor plan graph and other constraints via depth-first search.

4.3. Objective function for floor plan optimization

The objective function is defined on a floor plan where spaces are assigned to a node in the floor plan graph. The objective is composed of twelve constraints detailed as follows:

Shortest path to entrance constraint encourages unidirectional room access from the entrance. We compute the shortest path from all nodes to the floor's entrance, either the front entrance for the ground floor or the staircase for rooms upstairs. The path is computed in an axis-aligned fashion and can only traverse connected rooms on the floor plan graph. The amount of detour for each path is the percentage of the path in the wrong direction of the Euclidean distance from the entrance to that room. The objective function is computed as squared detours summed across rooms. Denote $\mathcal{F}$ as the set of all floors, e_f is the entrance on floor f, and $\to$ is the path allowed by the adjacency between rooms:

$$\mathcal{L}_{sp} = \sum_{f\in\mathcal{F}}\sum_{r\in f}\left(\frac{\|e_f \to r\|_{1,\text{direct}}}{\|e_f \to r\|_1} - 1\right)^2$$

Typical room area constraint encourages room of typical area so that the spaces serve the best function. A list of the typical area occupied by rooms is listed in Tab. 4, which is based on a typical US household. The ideal proportion of a room is computed as the typical area of that room divided by the sum of all the room's typical areas on that floor. The objective function is computed by the difference between a room's ideal proportion on one floor and the room's true proportion on that floor, summed across rooms. For all rooms $r \in f$, we compute its ideal area as

$$\overline{area}_r = \frac{typical\_area_r}{\sum_{r'\in f} typical\_area_r} area_f$$

A formula for the objective function is

$$\mathcal{L}_{ta} = \sum_{f\in\mathcal{F}}\sum_{r\in f} \max\left(\frac{\overline{area}_r}{area_r}, \frac{area_r}{\overline{area}_r}\right)$$

Room aspect ratio constraint encourages rooms of certain types to be square. The objective function is computed as the difference between the true aspect ratio and one, squared and summed across rooms. Denote by $\mathcal{R}_s$ the rooms needed to be square, we have

$$\mathcal{L}_{ar} = \sum_{f\in\mathcal{F}}\sum_{r\in f\cap\mathcal{R}_s}\left(\max\left(\frac{height_r}{width_r}, \frac{width_r}{height_r}\right) - 1\right)^2$$

Room type	Typical area
Kitchen	20
Bedroom	20
LivingRoom	25
DiningRoom	20
Closet	3
Bathroom	7
Utility	3
Garage	35
Balcony	6
Hallway	6
Staircase	20

Table 4. Typical area occupied by rooms

Room convexity constraint encourages rooms to be overall convex. The convexity of each room is computed as the ratio between the area of the convex hull of a room and the area of the room itself. The objective function is computed as the squared difference between the convexity of a room and one, summed across rooms.

$$\mathcal{L}_{conv} = \sum_{f \in \mathcal{F}} \sum_{r \in f} \left(\frac{area_{convex\_hull(r)}}{area_r} - 1 \right)^2$$

Room wall conciseness constraint encourages rooms to have fewer boundary edges, which allows rooms to have better-formed geometry along many iterations of perturbations. The objective function is the squared difference between the number of boundary edges of a room with four (minimal number of edges), summed across rooms.

$$\mathcal{L}_{conc} = \sum_{f \in \mathcal{F}} \sum_{r \in f} (\| w \in walls_r \| - 4)^2$$

Functional Room area constraint incentivizes the available area useful for dwellings of the people inside the house, characterized by functional rooms. Functional rooms include kitchens, bedrooms, living rooms, bathrooms, and dining rooms. The objective function is computed as the proportion of the area covered by these rooms, measured in squared distance with one. Denote by $\mathcal{R}_f$ the set of functional rooms, we have

$$\mathcal{L}_{func} = \sum_{f \in \mathcal{F}} \left(\frac{\sum_{r \in f \cap \mathcal{R}_f} area_r}{area_f} - 1 \right)^2$$

Room collinearity constraint incentivizes walls of multiple rooms to be collinear for aesthetics and construction purposes. The objective function measures the number of distinct X or Y coordinates for all walls of rooms across one floor.

$$\mathcal{L}_{col} = \sum_{f \in \mathcal{F}} (|\{x | \exists r \in f, x \text{ the x-coords of a wall in } r\}| \\ |\{y | \exists r \in f, y \text{ the y-coords of a wall in } r\}|)$$

Narrow passages constraint limits the number of passages in a room (including hallways) where people or furniture may find it hard to move across. We identify a narrow passage in a room by eroding and then buffering the 2D room contour with a certain threshold margin. Narrow passages inside a room are no longer present in the room contour after that erosion-buffer operation. The objective function is measured as the difference between the area of the room contour pre- and post- erosion-buffer operation. An illustration of the erosion-buffer operation can be found in Fig. 6. The formula can be written as

$$\mathcal{L}_{nar} = \sum_{f \in \mathcal{F}} \sum_{r \in f} \left(area_r - area_{\text{erosion-buffer}(r)} \right)$$

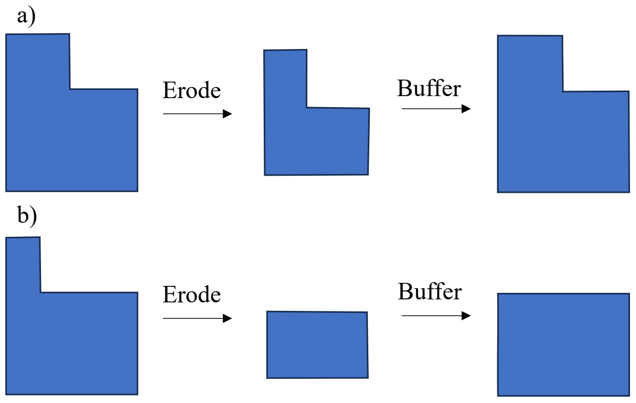


Figure 6. Illustration of narrow passage and erosion-buffer operation. a) In rooms with no narrow passage, the room contour is restored after the operation; b) In rooms with narrow passage, the narrow passage disappears from the room contour after the operation.

Exterior length by room constraint encourages rooms of certain types to cover most of the exterior walls and windows since people would expect more views of outside in these rooms and more privacy concerns in other rooms. The exterior room types include bedrooms and balconies, denoted by $\mathcal{R}_e$. The objective function is evaluated using the exterior length covered by these rooms divided by the exterior length covered by all rooms on that floor, measured by its squared distance with one.

$$\mathcal{L}_{extr} = \sum_{f \in \mathcal{F}} \left(\frac{\sum_{r \in f \cap \mathcal{R}_e} \sum_{w \in walls_r \text{ exterior}} \|w\|}{\sum_{r \in f} \sum_{w \in walls_r \text{ exterior}} \|w\|} - 1 \right)^2$$

Exterior corner by room constraint encourages the aforementioned room types to cover most exterior corners, which supposedly have better views. The objective function is measured by the percentage of corners covered by these rooms, measured by its squared distance with one.

$$\mathcal{L}_{extc} = \sum_{f \in \mathcal{F}} \left(\frac{\sum_{r \in f \cap \mathcal{R}_e} |\{c \in corners_r | c \text{ exterior}\}|}{\sum_{r \in f} |\{c \in corners_r | c \text{ exterior}\}|} - 1 \right)^2$$

Staircase occupancy constraint encourages the room assigned as the staircase room to cover the staircase placeholder space. It is measured as the percentage of staircase placeholder space covered by the staircase room, measured by its squared distance with one. Denote by sp the staircase placeholder and $\mathcal{R}_s$ the staircase rooms, we have

$$\mathcal{L}_{stair\_occ} = \sum_{f \in \mathcal{F}} \left(\sum_{r \in f \cap \mathcal{R}_s} \frac{area_{sp \cap r}}{area_{sp}} - 1 \right)^2$$

Staircase IOU constraint further encourages the room assigned as the staircase room to be exactly the same size, shape, and location as the staircase placeholder space. It is measured as the IOU of the staircase placeholder with the staircase, measured by its squared distance with one.

$$\mathcal{L}_{stair\_occ} = \sum_{f \in \mathcal{F}} \left(\sum_{r \in f \cap \mathcal{R}_s} IOU_{r,sp} - 1 \right)^2$$

4.4. Floor plan optimization moves

While solving for the aforementioned constraints, we need to design a set of moves to perturb the floor plan, which are listed as follows and illustrated in Fig. 7:

Extruding a wall segment inwards randomly select one wall segment of a room in the current floor plan and move it towards the inside of the room by one grid size (0.5). Other rooms sharing part of the wall with the selected wall will fill up the space left by the move.

Extruding a wall segment outwards randomly select one wall segment of a room in the current floor plan and move it towards the outside of the room by one grid size (0.5). Other rooms sharing part of the wall with the selected will give up their space to the room.

Swapping the assignment for adjacent rooms randomly select one space for a room and its neighbor and swap their room assignment.

In all of the above moves, we reject moves that lead to a floor plan that does not suffice the floor plan graph on that floor. We also reject moves that lead to invalid geometry, including degenerate, disconnected, or out-of-boundary rooms, and those that fail to satisfy the constraints on exterior rooms and staircase placeholders. One may think of satisfying floor plan graphs as a hard constraint.

Moving staircase. We also provide an additional move for the staircase placeholder. The staircase placeholder can move along one of the axes by one grid size.

At each iteration of the simulated annealing, we first select one of the floors to operate on or choose to move the staircase placeholder. Then, we randomly choose one of the three moves to apply. A move is rejected if it no longer satisfies the hard constraint given by the floor plan graph or rejected by the simulated annealing probability computed using the change in the objective function.

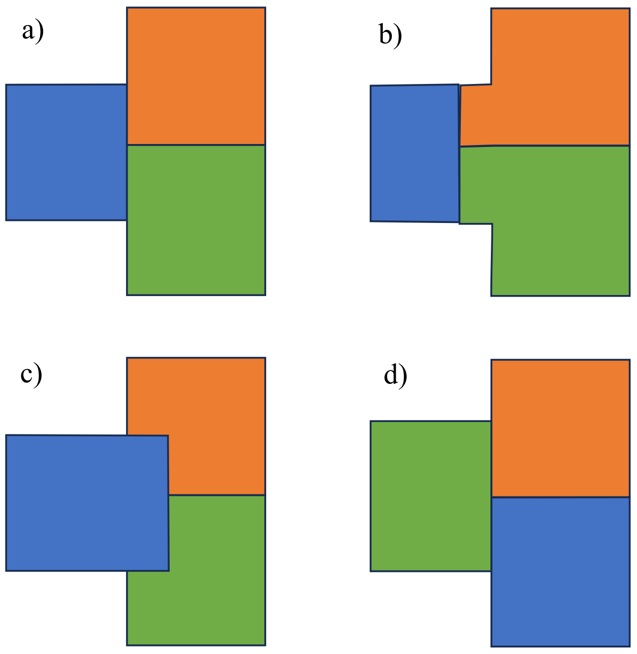


Figure 7. All floor plan optimization moves. a) The original floor plan, with each color showing the assignment of each room (e.g. blue for *living-room_0*, orange for *bedroom_0*, and green for *bedroom_1*; b) floor plan after extruding rightmost wall segment of the blue room inwards; c) floor plan after extruding rightmost wall segment of the blue room outwards; d) floor plan after swapping the assignment for the green and the blue room.

4.5. Postprocessing of floor plan

After acquiring the floor plan for all floors, we must convert it to a mesh. Each space assigned to a room is extruded by the height of the wall and solidified by the thickness of the wall; both parameters are the same across all rooms. Besides placing furniture, we conduct the following postprocessing operations.

Placement of doors and windows. For pairs of rooms that share an edge in the floor plan, they must share a wall segment with its length over a certain threshold. We then cut the shape of the door from both room meshes and place the door in that space. Doors can be opened towards the inside of the house or away from the house's entrance for ergonomics. For other pairs of rooms designated by the user, i.e., between dining rooms and living rooms, one may choose to remove all the walls in between and place no doors. For rooms facing the exterior of the house, if they can have windows installed, we selectively cut off the shape of the window from the room meshes with a limit on the maximal width of the window. Then, we place windows in these shapes. Landscapes are placed outside the window.

Adding materials to floors, ceilings, and walls. The walls of rooms are applied with the following materials conditioned on the room type: (ceramic) square tile, concrete, brick, or plaster. The floors of rooms are applied with the following materials conditioned on the room type: tiled wood floors, square or hexagonal, alternating or non-alternating tiles, rug or concrete. The ceilings of rooms are applied with plaster material. Materials are sometimes shared across different rooms.

Adding staircases. We compute the intersection of the space assigned as staircase rooms on consecutive floors. Our constraint solver will make sure that the intersection is at least the size of the staircase placeholder, which is non-empty. We randomly sample one staircase per floor (excluding the topmost one) and position them inside their corresponding staircase rooms. We reject samples where the staircase and the room in front of the steps fall outside the room or when the consecutive staircase intersects. We cut off the shape of the stairs from the room meshes and add guard rails around the stairs.

5. Constraint Solver Details

5.1. Greedy Solving Algorithm

Optimizing over all rooms and all objects at the same time is unfeasible due to the magnitude of the state-space. As a result, we use a greedy algorithm to first solve the floor-plan, then solve large, medium, and small objects respectively. At

Algorithm 1 Greedy Solving Algorithm

```
1: procedure GREEDYSOLVER($P$)
2:     SIMULATEDANNEALING($P, Rooms, r$)
3:     for $r$ in $rooms$ do
4:         SIMULATEDANNEALING($P, BigObjects, r$)
5:         SIMULATEDANNEALING($P, MediumObjects, r$)
6:         SIMULATEDANNEALING($P, SmallObjects, r$)
7:     end for
```

each stage we solve each room separately. A very high-level pseudocode of our solver algorithm is given as Algorithm 1. This algorithm is not optimal in any sense, but the problem at hand is computational intractible, and an optimal solution is not required to obtain aesthetically pleasing scenes. We provide this solver to prove our language can be optimized efficiently, and to serve as a baseline for future improvements or followup work.

5.2. Move Utilities

Cardinality Bounding Our solver starts with an empty scene, and must add objects during optimization to satisfy object-quantity constraints and objectives given by the user. We implement this via the *Addition* and *Deletion* moves described below, which add or remove one object. Choosing to propose a random object type with a random set of relations would have vanishingly small likelihood of producing a move which obeys the given constraints - typically only a few object types and a few relation assignments (against wall, on floor, etc) are actually valid.

To optimize efficiently, we implemented a recursive procedure to traverse through the constraint graph and find every relevant context (such as ”ontop of bookcase” or ”against livingroom wall”) available in the current scene state, retrieve any lower/upper bounds on object counts to be placed into these contexts. For example if there are two shelves in the current state, and the user has specified each shelf shall have between 1 and 5 books placed on it, our procedure would return 2 bounds, one for each shelf, with 1 and 5 as the lower and upper bounds on object count.

These bounds are sensitive to the current state of the scene: if the users specifies there should be more chairs in the dining room than tables in the dining room, then the current number of chairs will be used as an upper bound for the number of tables and vice versa. This allows optimization of arbitrary inequalities between object counts, since by randomly performing valid additions and deletions, the optimizer will explore the full space of discrete object counts for every possible context.

Degree Of Freedom Computation Moving objects in the full 6D pose space is completely unfeasible because of the plane assignment hard constraints that need to be satisfied.

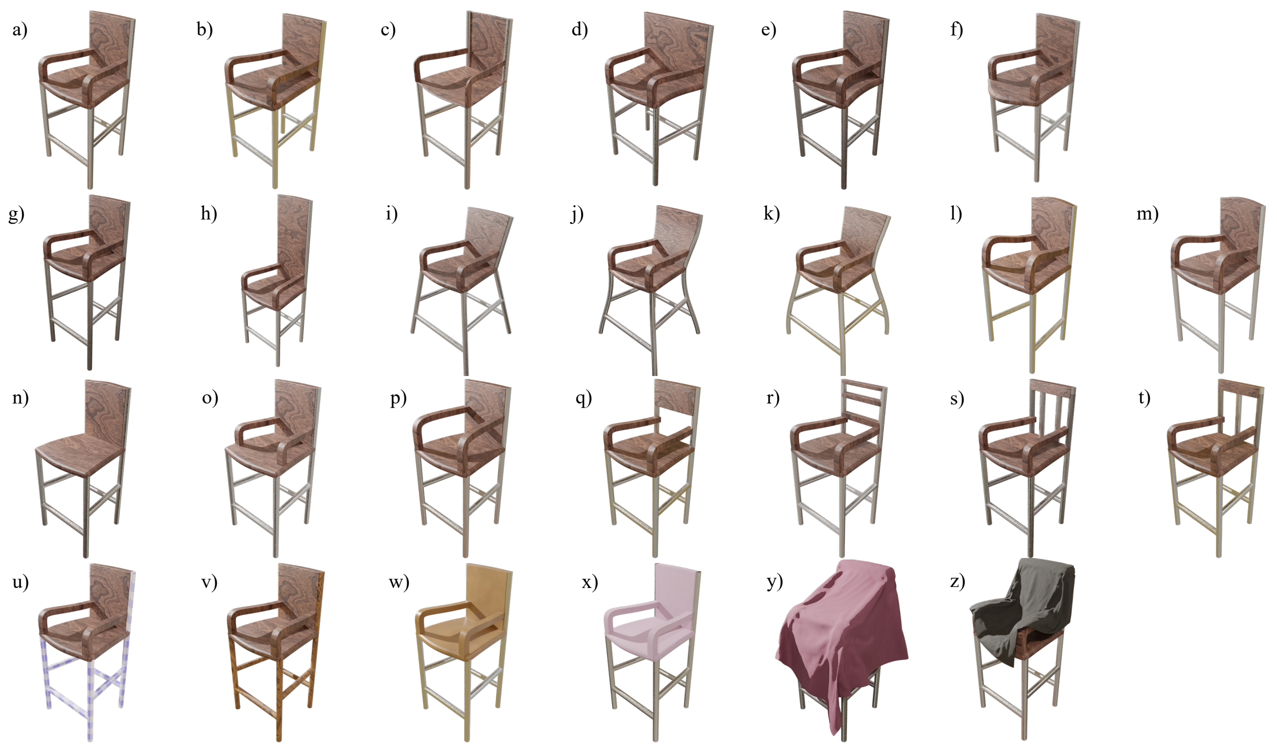


Figure 8. Variation in a chair asset with tuneable parameters. a) A base chair for comparison, followed by chairs with b) larger depth; c) thinner seats; d) wider backs; e) more curvature in seats; f) extrusion in front; g) longer legs; h) larger backs; i)-k) tilted / outward-bending / inward-bending legs; l), m) no leg bars along both axes; n) no arms; o)-p) arms with different attachments to the seat and back; q)-t) backs partially covered/supported by horizontal or vertical bars; u)-v) different leg material (woven fabric / wood); w-x) different seat material (leather / fabric); y)-z) different placement of blankets.

Enforcing these constraints by minimum distance scores and considering movement only on the XY plane is another option, but this still causes violation of the hard constraints and is also wasteful as an optimizer state-space. Therefore, we calculate the degrees of freedom of each object and only move objects along the allowed subspace (e.g. painting only moves on the wall it is assigned to).

For each object, we first obtain the planes that the object is constrained to move on. We then compute two types of DOFs. The translation DOFs are computed as the matrix of projection onto the intersection subspace of the planes. If the constraints are contradictory, this will be the zero matrix. The rotation DOF is either the free axis of rotation around which the object is allowed to rotate, or none if the constraints do not allow rotational movement.

Resolving Discrete Move Poses The optimizer needs to initialize every object that is added to the scene before proposing any moves to it. This initialization must obey the plane assignment hard constraints, so that the subsequent continuous moves also obey the hard constraints. Thus, we initialize objects by essentially sampling a random position on the subspace defined by the plane assignments and sampling a random rotation that is a multiple of $\pi/2$. The position sampling is done by sampling a random position on the first plane, and then repeatedly snapping the object to its assigned planes with the specified margin. The validity of the initialization is checked after each attempt, and if each initialization attempt is unsuccessful for a certain number of attempts (20 by default), the initialization is unsuccessful, and the move is reverted.

Reversing Moves Not every proposed move is a valid move. For instance, translating a painting too much might cause it to overhang, or reassigning a sofa to another wall might make it intersect another object in the scene. As a result, after we apply any move, we check its validity. This is done by checking that the chosen object does not collide with any other mesh, and all the relation constraints of the object are satisfied. If the move is not valid, then we reverse the move to remove its effects. For instance, if the object was moved by a rotation or translation, and the resulting state is not valid, we restore the backup pose of the object. If the move was an addition, we remove the object, and so on.

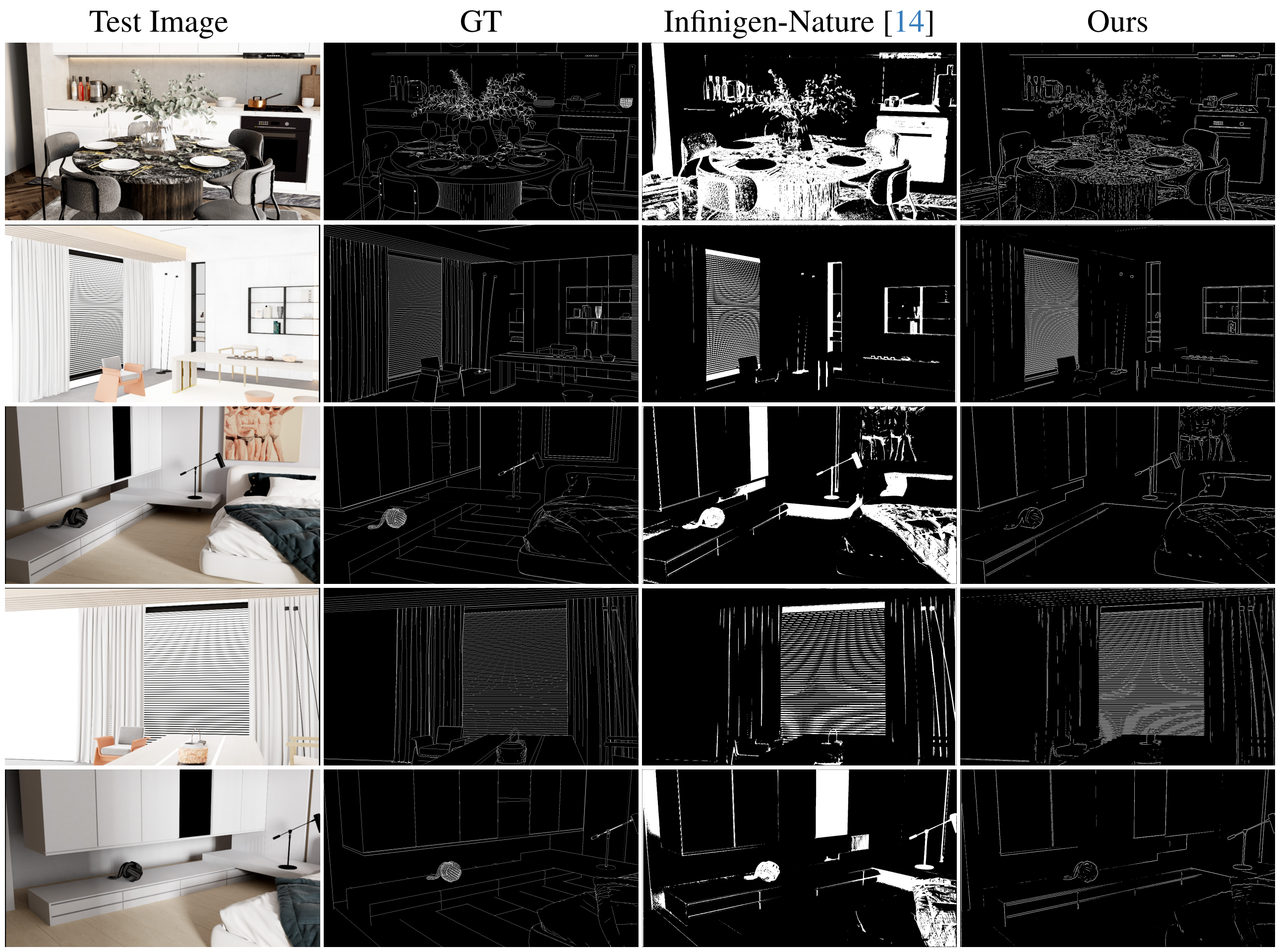


Figure 9. Additional qualitative results on synthetic scenes [8].

5.3. Move Implementations

Addition To perform an addition, we extract all available cardinality bounds from the current. Then, we discard all bounds which are tight above, i.e. those for which adding an object would violate an upper bounds.

The most challenging stage of addition is finding a satisfying assignment for relevant constraints. If the user specified `scene[Seating].related_to(room, on_floor).related_to(room, against_wall).count() > 0` then we know we must add some kind of seating that is against both the floor and wall. However, many options exist: Seating could mean either an armchair or a sofa, there are many possible floors to place the seating onto, and many possible wall planes attached to each floor plane. Moreover, any choice for these variables could activate additional constraints, for example the user may have written a rule that applies to all sofas in the scene, or all objects in a particular room, so if we choose for our seating object to be a sofa, or if we choose to put it in that particular room, then additional constraints may be added to the list yet to be satisfied.

This relation assignment problem is related to classic SAT solving, except for that making an assignment can add additional terms to the equation. Or equivalently, it is a SAT problem where the full equation to be satisfied is deceptively long due to new constraints becoming activated. We anticipate that future versions of our solver can incorporate classic SAT solving approaches directly. However, for our current constraint programs we have found it is sufficient to perform exponential search over all options, visiting each child node in the search tree in a random order to ensure unbiased results. This approach is exponential in the number of semantic and relationship constraints involved, but fortunately these rarely number more than 3 or 4 (IE, 1-2 semantic classes) and the branching factor tends to be small (IE, relatively few different specific object options, or few different wall planes to assign to).

For each valid assignment found, we procedurally generate a ”placeholder” asset and attempt to fit it into the scene as described in ”Resolving Discrete Moves” as described above.

Placeholders are special versions of our 3D assets provided by each procedural generator which have mostly planar surfaces and lower polygon count. E.g. the placeholder for a chair would still have properly shaped legs, seat and backrest, but would not have any bevels, chamfers, nails/screws or fine geometric details. This lower polygon representation speeds up collision checking, and eliminates the need for us to heuristically detect flat planes on the object, since the asset author provides these procedurally.

Deletion Deletion uses the same cardinality bound logic as addition, but chooses a random object cardinality bound that is not tight below, and proposes to delete it to see if score is reduced. Typically these moves do not help the immediate score, as we incentives placing as many objects as possible, but can help to eliminate particularly poorly placed objects or to avoid local optima in object counts.

Resample Resample's primary function is to replace an existing object in the scene with an object of the same class but with new parameters. This often causes a change in shape, e.g. the length/width of a table will change, or the number of cells in a shelf may increase. Changing these parameters is desirable as it may increase/decrease the objective function (e.g. if a volume() or min_distance) is changed as a result). To place the new object in the scene, we try aligning each of the bottom corners of the new bounding box with that of the old object, and check each pose for collisions, which allows the object to grow/shrink strictly to the left or right if it is attached to a wall. We assume relation assignments from the old object remain valid, since regenerating an object with new parameters does not change its semantics.

Translate Let P be the projection matrix computed as the translational degree of freedom for the chosen object. We sample $\vec{x} \in \mathbb{R}^3$ where $x_i \sim \mathcal{N}(0, \sigma^2)$ for $i = 1, 2, 3$, and the variance σ^2 is proportional to the temperature. The object is then translated by $\mathrm{P}\vec{x}$. This makes the object take a random step along the subspace on which it is constrained to.

Rotate Let $\vec{e}$ be the axis of rotation computed as the rotational degree of freedom for the chosen object. We sample $\theta \sim \mathcal{N}(0, \sigma^2)$ where the variance σ^2 is proportional to the temperature. The object is then rotated by θ around the axis $\vec{e}$. This makes the object take a small random rotation on the subspace on which it is constrained to.

ReinitPose reinitializes the 6DOF pose of the object by resolving the discrete move poses again. Since the object relations are the same, the effect is essentially sampling a random position and orientation on the same constraint subspace. This move is useful for getting a good layout in the early stages of optimization and the cases in which an object is stuck in a sub-optimal position.

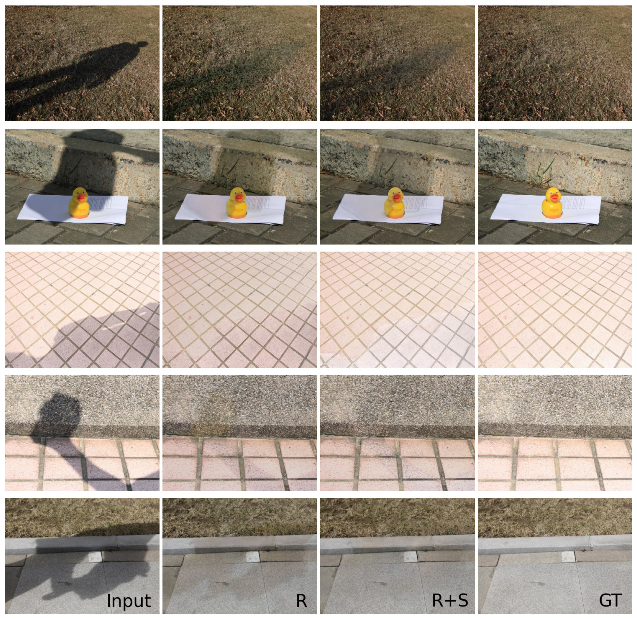


Figure 10. Additional qualitative zero-shot results on SRD [13] test dataset.

ReassignPlane . As explained in Addition, if the user specifies an object to be placed against one or more surface(s), then multiple options usually exist for which surfaces to use. This move simply attempts to swap the object to a different plane, e.g. move a sofa to a different wall, or a bottle to a different row of a shelf.

ReassignTarget Similarly, multiple options often exist for which object an object is a child of in the scene graph, e.g. a plant pot could rest on one of many different shelves/tables in a room. This move swaps the object to be a child of some other object in the scene which satisfies the same constraints as its current assignment.

6. Asset Generation Details

6.1. Asset Coverage and Variation

We provide 79 randomized procedural object generators. By category, we cover Appliances (10 generators, 112 params), Windows/Doors/Staircases (14 generators, 127 params), Furniture (17 generators, 216 params), Decorations (15 generators, 92 params) and Small Objects (19 generators, 194 params). We provide 30 material generators, 120 params total, split approximately evenly between types of Wood, Ceramic, Fabric, Metal and others. Materials are assigned to objects via customizable weighted lists, e.g. spatulas invoke either wood, plastic or metal generators for each of their ends. Following Infinigen, we report procedural parameter count as a proxy of complexity; each parameter is a random but controllable degree of freedom e.g "number of seats on a sofa". In all, we provide 40k lines of code, of which 25k are object/material generators. For asset coverage, an incomplete list of the assets we cover is listed in Tab. 6. For asset variations, an illustration of the variation of assets is shown in Fig. 8.

6.2. Lighting and Camera Placement

Lights and windows use similar constraints as other objects (e.g. maximize count & spacing), with random wattage/temperature sampled from real-world ranges. Camera selection follows Infinigen [14]: we sample at random, reject near walls, and maximize depth variance.

7. Experiment Details

7.1. Shadow Removal

We use the model implementation from [9]'s codebase to train the two variants of the model: R (trained on real dataset only) and R+S (trained on real and synthetic datasets). Since the codebase lacked a validation set, we developed our own, comprising all image pairs across four scenes from the ISTD training dataset. Additionally, in contrast to the provided implementation, we used an L1 loss as stated in the paper. We trained the two variants for 30k steps each, including a 30-epoch linear warmup phase, using AdamW [11] optimizer with default hyperparameters and a learning rate of $2e-4$. We chose the runs to have an effective batch size of 32 (by accumulating gradients for 4 steps and using the actual batch size to be 8). The training process utilized four Nvidia 3090 GPUs, with Mixed-16 precision. Figure 10 shows additional qualitative results.

We opted not to use the pretrained model from the codebase, as our attempts to reproduce the results were unsuccessful. Nevertheless, to ensure a fair comparison, we adhered to the same implementation details for both variants. Additionally, we chose not to report SSIM (Structural Similarity Index Measure), since both models demonstrated equivalent performance, with no significant difference observed when rounding to two decimal places, for this metric.

7.2. Occlusion Boundaries

We separately train three U-Net [17] models from scratch on images generated from Infinigen Indoors, Infinigen [14] and Hypersim [16]. We apply random {cropping, brightness contrast} and color jittering with probability 0.6. We also use the RMSprop optimizer with a base learning rate of 10^{-5}, a momentum of 0.99 and a weight decay of 10^{-8}. Each model is trained for 10 epochs using binary cross entropy loss.

Due to the absence of ground truth occlusion boundaries in Hypersim (or any other photorealistic dataset), we approximate them by thresholding the gradient of the provided depth maps. We carefully tuned this threshold on Hypersim to give the best results.

We compare the performance of the U-Net models on a curated test set of photo-realistic artist-designed synthetic 3D scenes for architecture visualization [8]. We extract the ground truth occlusion boundaries of these scenes using the tools provided in Infinigen. Additional qualitative results shown in Fig. 9 underscore our claim that the Infinigen Indoors - trained model generalizes better.

Method	Mean Error Frequency ↓	More ↑ Realistic	More Realistic Layout ↑	Realism CI 99%	Layout Realism CI 99%
ProcTHOR [3]	0.252	0.107	0.056	$[0.054, 0.187]$	$[0.021, 0.127]$
ATISS [12]	0.232 [12]	0.287	0.307	$[0.198, 0.389]$	$[0.217, 0.410]$
SceneFormer [19]	0.713 [12]	0.333	0.440	$[0.241, 0.439]$	$[0.339, 0.547]$
FastSynth [15]	0.414 [12]	0.093	0.147	$[0.046, 0.171]$	$[0.083, 0.234]$
Ours	**0.175**	**0.795**	**0.760**	$[0.750, 0.835]$	$[0.712, 0.803]$

Table 5. **Perceptual Study Results**. We followed the method and metrics from ATISS, but added *Layout Realism*, which says to only consider arrangement. We used each method's default renderer.

8. Perceptual Study

Following ATISS [12], we conducted a perceptual study on Amazon Mechanical Turk to evaluate the realism of the generated scenes and the realism of the generated layouts. We compared Infinigen Indoors to ProcTHOR [3], ATISS [12], SceneFormer [19], and FastSynth [15]. We presented the subjects pairs of images from each method (for instance Infinigen vs ProcThor) to evaluate overall realism and layout realism. For mean error frequency, we asked the subjects if the image from a method contained any obvious errors such as flying furniture, overlapping furniture, etc. For layout realism, we asked the subjects to focus only on the arrangement of the furniture and ignore the style of individual objects. Table 5 shows that the subjects preferred Infinigen Indoors over all methods in terms of both realism, layout realism, and the lack of obvious errors. An important caveat is that "realism" may be influenced by asset and lighting quality.

- Household appliances
  - Fridge, Beverage fridge (with racks)
  - Dishwasher (with racks)
  - Microwave
  - Oven, Stove, Oven with stove (with racks)
  - TV, Monitor
  - Kitchen Sink (with faucets)
- Bathroom fixtures
  - Bathroom sink (standing / embedded / tabletop)
  - Bathtub (alcove / freestanding / corner)
  - Hardware (towel bar / towel ring / toilet roll paper holder / robe hooks)
  - Toilet (two-piece / one-piece / in-wall)
- Clothes
  - Pants (underwear / shorts / pants)
  - Shirts (T-shirts / shirts)
  - Blankets / towel (folded / rolled)
- Architectural Elements
  - Doors
    - Lite / Louver / panel / glass panel door
    - Door casings
  - Staircases (with treads / banisters / guardrails / glass railings)
    - Straight / Cantilever / L-shaped / U-shaped staircase
    - Spiral / curved staircase
  - Rugs
  - Warehouse racks / pallets
- Seatings
  - Bar stool / office chair
  - Armchair / dining chair / side chair / spholstered chair
  - Beds (bedframe / mattress / pillow)
  - Sofa
- Shelves (with drawers and doors)
  - Cabinets / kitchen cabinets
  - Cell shelves / wall shelves / bookcases / triangle shelves
- Table decorations
  - Books (column / stack)
  - Vases / Aquarium tank
  - Plants in pots (floor-top / table-top)
- Tables
  - Desks / Cocktail table / Dining table / Kitchen table
- Tableware
  - Bottle (soda / wine / beer / juice) / jar
  - Chopsticks / Knife(table knife / cleaver / chef's knife) / forks / spoons / spatulas / bowl / plate
  - Cup (mug / shot glass / teacup / plastic cup) / wineglass
  - Food bag(chip bag / food pouch / food bar) / Food box / can / jar / Fruits in containers(i.e. tableware with fruits placed inside)
  - Pan / pot(Cooking pot / saucepan) / lid(of pots and pans)
- Wall decorations
  - Balloons / wall arts / mirror
- Windows
  - Sliding / awning / casement / glassblock / bay window

Table 6. Coverage of the assets in Infinigen Indoors.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. 7

[2] David F. Crouse. On implementing 2D rectangular assignment algorithms. *IEEE Transactions on Aerospace and Electronic Systems*, 52(4):1679–1696, 2016. Conference Name: IEEE Transactions on Aerospace and Electronic Systems. 6

[3] Matt Deitke, Eli VanderBilt, Alvaro Herrasti, Luca Weihs, Jordi Salvador, Kiana Ehsani, Winson Han, Eric Kolve, Ali Farhadi, Aniruddha Kembhavi, and Roozbeh Mottaghi. Procthor: Large-scale embodied ai using procedural generation. *ArXiv*, abs/2206.06994, 2022. 17

[4] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016. 7

[5] Matthew Fisher, Daniel Ritchie, Manolis Savva, Thomas Funkhouser, and Pat Hanrahan. Example-based synthesis of 3D object arrangements. *ACM Transactions on Graphics*, 31 (6):135:1–135:11, 2012. 1

[6] Frank M. Frey, Aaron Robertson, and Michael Bukoski. A method for quantifying rotational symmetry. *New Phytologist*, 175(4):785–791, 2007. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1469-8137.2007.02146.x. 6

[7] Qiang Fu, Xiaowu Chen, Xiaotian Wang, Sijia Wen, Bin Zhou, and Hongbo Fu. Adaptive synthesis of indoor scenes via activity-associated object relation graphs. *ACM Transactions on Graphics*, 36(6):201:1–201:13, 2017. 1

[8] Gumroad. Gumroad. `https://discover.gumroad.com/`. 15, 17

[9] Lanqing Guo, Siyu Huang, Dingshuo Liu, Hao Cheng, and Bihan Wen. Shadowformer: Global context helps image shadow removal. *ArXiv*, abs/2302.01650, 2023. 17

[10] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987. 6

[11] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2017. 17

[12] Despoina Paschalidou, Amlan Kar, Maria Shugrina, Karsten Kreis, Andreas Geiger, and Sanja Fidler. Atiss: Autoregressive transformers for indoor scene synthesis. In *Advances in Neural Information Processing Systems*, pages 12013–12026. Curran Associates, Inc., 2021. 17

[13] Liangqiong Qu, Jiandong Tian, Shengfeng He, Yandong Tang, and Rynson W. H. Lau. Deshadownet: A multi-context em-

bedding deep network for shadow removal. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2308–2316, 2017. 16

[14] Alexander Raistrick, Lahav Lipson, Zeyu Ma, Lingjie Mei, Mingzhe Wang, Yiming Zuo, Karhan Kayan, Hongyu Wen, Beining Han, Yihan Wang, Alejandro Newell, Hei Law, Ankit Goyal, Kaiyu Yang, and Jia Deng. Infinite photorealistic worlds using procedural generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12630–12641, 2023. 15, 17

[15] Daniel Ritchie, Kai Wang, and Yu-an Lin. Fast and Flexible Indoor Scene Synthesis via Deep Convolutional Generative Models, 2018. arXiv:1811.12463 [cs]. 17

[16] Mike Roberts, Jason Ramapuram, Anurag Ranjan, Atulit Kumar, Miguel Angel Bautista, Nathan Paczan, Russ Webb, and Joshua M. Susskind. Hypersim: A photorealistic synthetic dataset for holistic indoor scene understanding. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 10912–10922, 2021. 17

[17] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, pages 234–241. Springer, 2015. 17

[18] Daniel M. Roy and Yee Whye Teh. The mondrian process. In *Neural Information Processing Systems*, 2008. 10

[19] Xinpeng Wang, Chandan Yeshwanth, and Matthias Nießner. Sceneformer: Indoor scene generation with transformers. *2021 International Conference on 3D Vision (3DV)*, pages 106–115, 2020. 17

[20] Lap-Fai Craig Yu, Sai-Kit Yeung, Chi-Keung Tang, Demetri Terzopoulos, Tony F. Chan, and S. Osher. Make it home: automatic optimization of furniture arrangement. *ACM SIGGRAPH 2011 papers*, 2011. 1, 4

[21] Hagit Zabrodsky, Shmuel Peleg, and David Avnir. Continuous symmetry measures. *Journal of the American Chemical Society*, 114(20):7843–7851, 1992. Publisher: American Chemical Society. 6