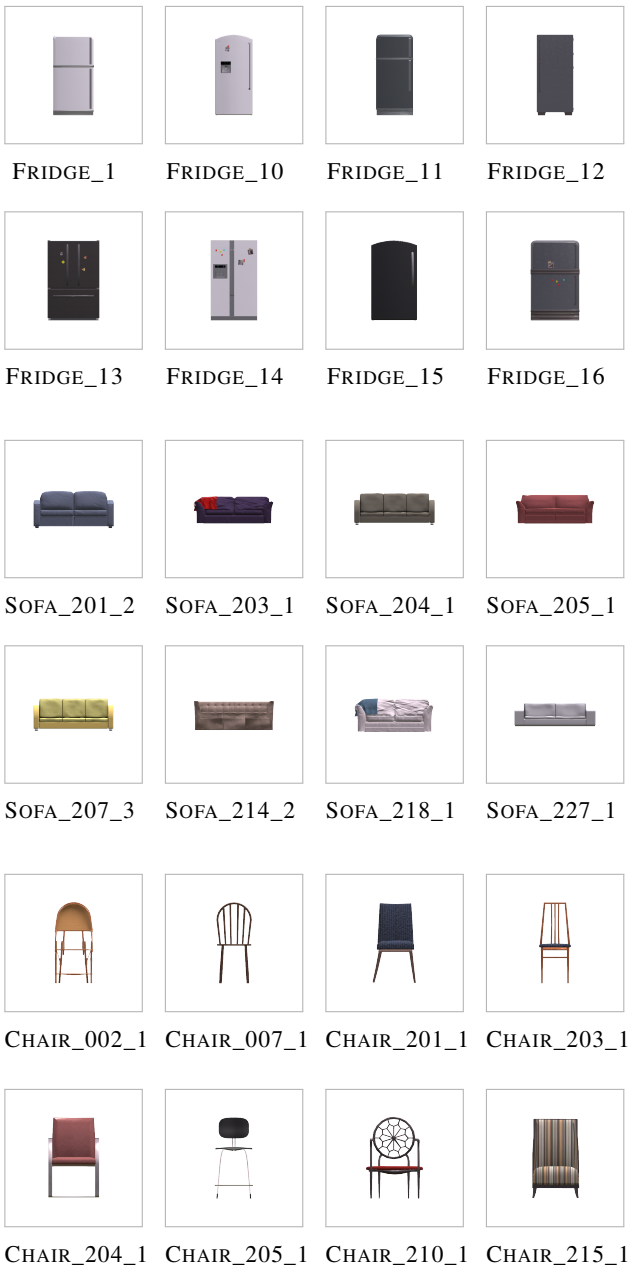
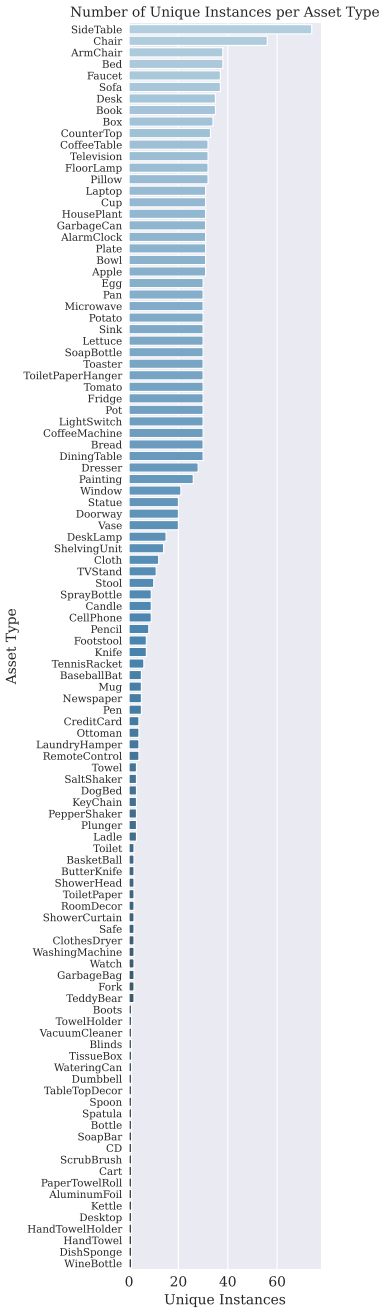
Appendix

A ProcTHOR Assets



(a) Examples of assets in the asset database. The forward-facing direction for each asset is consistent across all assets within its type, which allows us to do things like not spawn fridges facing the wall.



(b) The number of unique 3D modeled assets for each of the 108 asset types. There are 1,633 unique assets in total.

Figure 1: Examples and statistics of assets in the asset database.

B House Generation

This section gives more details about the process we developed to procedurally sample houses.

B.1 Examples

B.1.1 3-Room Houses

Figure 2: Examples of 3-room houses generated in PROCTHOR-10K.

B.1.2 4-Room Houses

Figure 3: Examples of 4-room houses generated in PROCTHOR-10K.

B.1.3 5-Room Houses

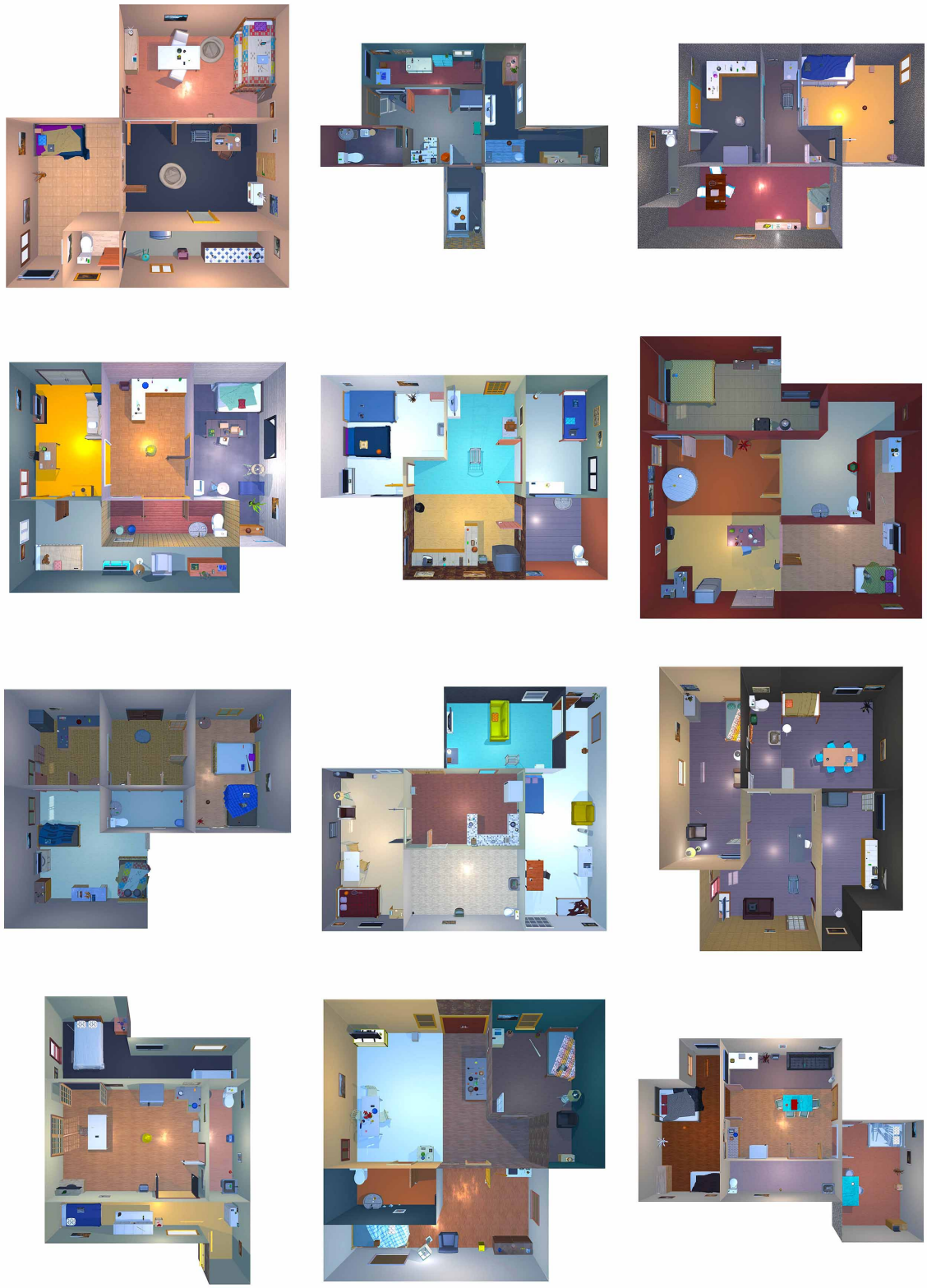


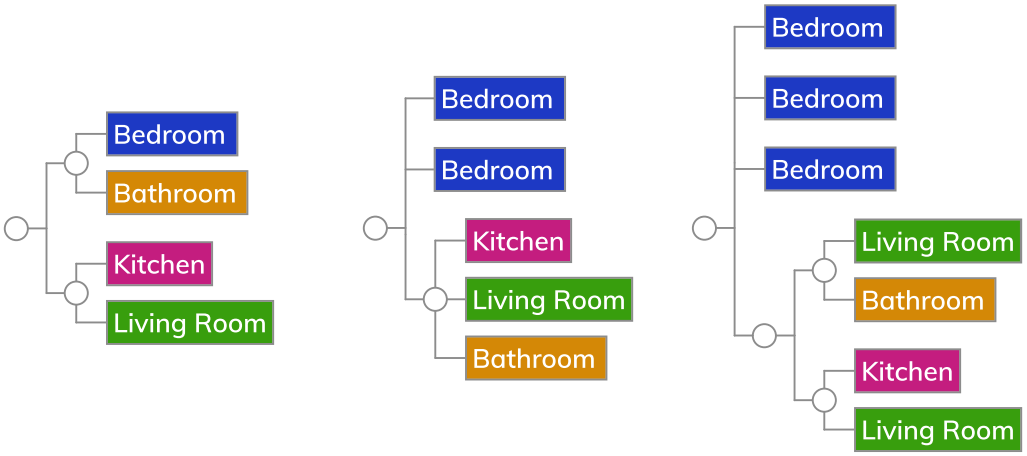
Figure 4: Examples of 5-room houses generated in PROCTHOR-10K.

B.1.4 6-Room Houses

Figure 5: Examples of 6-room houses generated in PROCTHOR-10K.

B.1.5 7+ Room Houses

Figure 6: Examples of 7+ room houses generated in PROCTHOR-10K.



(a) 4-Room House (b) 5-Room House (c) 7-Room House

Figure 7: Examples of room spec hierarchies used to sample differently sized houses.

B.2 Room Specs

Room specs provide the ability to specify the rooms that appear in a house, the relative size of each room, and how the rooms are connected with doors. Their idea was first proposed in [65]. A room spec is manually specified with a tree data structure.

Figure 7a shows a simplified example of a room spec with four rooms: bedroom, bathroom, kitchen, and living room. In this room spec, there are two subtrees, comprising $\mathcal{Z}_{bb} = \{\text{bedroom, bathroom}\}$ and $\mathcal{Z}_{klv} = \{\text{kitchen, living room}\}$. At each level of the tree, there is a constraint that there must be a direct path connecting every child node of a parent. Thus, in our example, there will be a path between the bedroom and the bathroom, a path between the kitchen and the living room, and another path connecting $\mathcal{Z}_{bb}$ to $\mathcal{Z}_{klv}$. We can also specify which room types we would prefer not to have a path between it and the parent. For example, we typically do not want the bathroom to have 2 doors, such as between it and the bedroom and between it and a room in $\mathcal{Z}_{klv}$.

Each tree node, below the root of the tree, is also assigned a growth weight, which approximates the relative size of the node compared to all other nodes that share the same parent. For instance, we might assign both $\mathcal{Z}_{bb}$ and $\mathcal{Z}_{klv}$ a growth rate of 1, to be roughly the same size. But, if we want the bedroom to take up roughly 60% of the $\mathcal{Z}_{bb}$'s area, then we might assign the bedroom a growth rate of 3 and the bathroom a growth rate of 2.

Room specs allow us to flexibly choose the distribution of houses we sample, allowing us to specify massive mansions, studio apartments, and anything in-between. Moreover, just a few room specs can go a long way. To generate our houses, we use 16 room specs, which each uses between 1 to 10 rooms. To generate the houses dataset, we assign a sampling weight to each of our room specs, and then use weighted sampling to sample a room spec for each house.

B.3 Sampling Floor Plans

The size and shape of the house are sampled to form the interior boundaries. Room specs specify the distribution over the dimensions of the house. Figure 8 visualizes the process of sampling an interior boundary, where we first sample the size of the boundary and then make cuts to the corners to add randomness. The sampling starts off by choosing the initial upper bound of the top-down x and z size of the house, in meters, respectively denoted as x_s and z_s. Each dimension is an integer. In most room specs, each dimension is independently sampled from the discrete uniform distribution $x_s, z_s \sim U(\max(\ell_{\min}, \mu_a\sqrt{n_r} - \mu_a/2), \mu_a\sqrt{n_r} + \mu_a/2)$, inclusive. However, individual room specs can override the x_s and z_s sampling distributions. Here, n_r represents the number of rooms in the house, $\ell_{\min}$ is set to 2 and represents the minimum size of x_s and z_s, and μ_a is set to 3 and represents the average size of x_s and z_s per room.

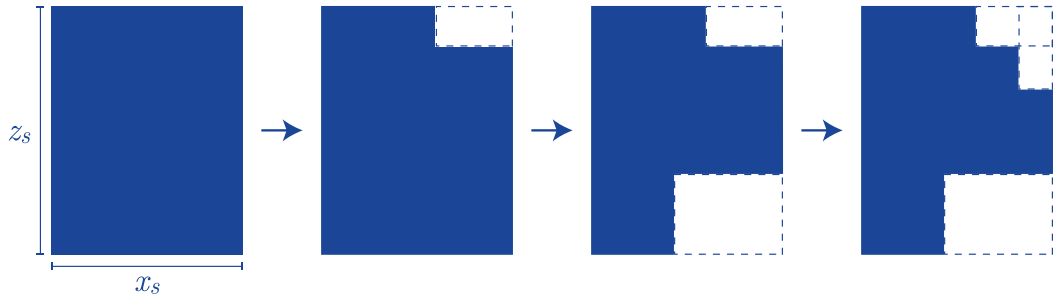


Figure 8: An example of the interior boundary cut algorithm. The images show a top-down view of the house's floor plan. First, we sample an interior boundary rectangle (x_s, z_s), which is shown on the left. Then, we make n_c rectangular cuts to the corners of the rectangle to make the interior boundary of the house a more complex polygon. In this case, we make $n_c = 3$ cuts to form the final interior boundary, which is shown on the right.

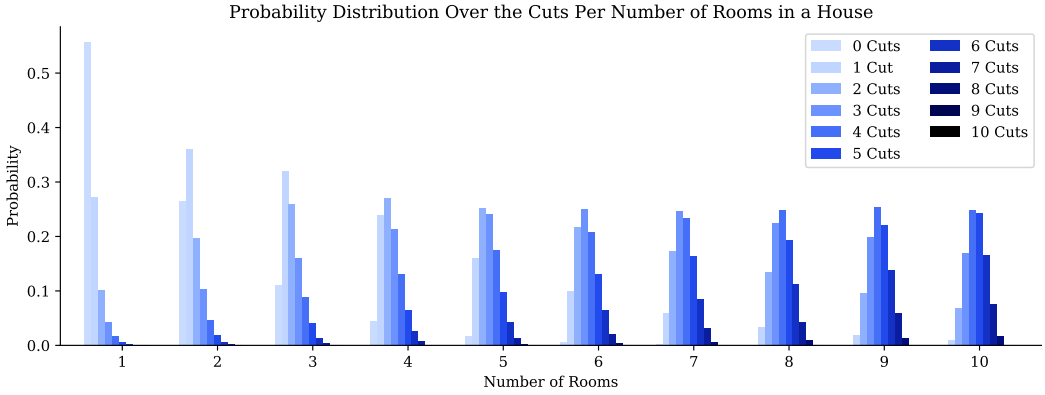


Figure 9: The probability distribution over the number of cuts, n_c, made to the rectangular boundary (x_s, z_s) with respect to the number of rooms in the house, n_r. Notice that when there are more rooms in the house, the number of cuts in the distribution increases.

Once we have the rectangular boundary (x_s, z_s), we then make several *cuts* to the outside of the rooms such that the interior boundaries can take on the shape of more complex polygons. The number of cuts, n_c, is sampled from the distribution $n_c \sim \lfloor 10 \cdot \text{Beta}(\alpha_c, \beta_c) + 1/2 \rfloor$, where $\alpha_c = n_r/2$ and $\beta_c = 6$. Figure 9 shows the distribution that is formed with respect to the number of rooms in the house, n_r. When there are more rooms, the probability distribution over the number of cuts increases. Since the range of the beta distribution is $(0, 1)$, the upper bound on the number of cuts is exactly 10.

The size of each cut is a rectangle, in meters, denoted by (c_x, c_z). Both c_x and c_z are sampled from integer distributions. We sample from $c_x \sim U(1, \max(2, \min(x_s - 1, \lfloor a_{\max}/2 \rfloor) - 1)$, inclusive, where $a_{\max}$ is set to 6 representing the maximum cut area. We then sample $c_z \sim U(1, a_{\max} - c_x)$. The position of where the cut happens is anchored to one of the 4 corners of the interior boundary, where the exact corner is independently and uniformly sampled each time.

Since the size of each cut is an integer, and the rectangular boundary sizes are also integers, we can efficiently represent the interior boundary with a (x_s, z_s) boolean matrix. Here, we could have 1s representing where the inside of the interior boundary and 0s representing the outside of the interior house boundary.

Given a room spec and an interior boundary, we use the algorithm proposed in [61] to divide the interior boundary into rooms. The algorithm recursively subdivides the interior boundary for each subtree in the room spec. Figure 10 shows an example using Figure 7a's room spec. The algorithm first divides the interior boundary into two zones, the "bedroom & bathroom" zone and the "kitchen & living room" zone. The "bedroom & bathroom" zone then subdivides into two rooms, the bedroom and bathroom. Similarly, the "kitchen & living room" zone is also subdivided into two rooms, the

Figure 10: An example of the recursive floor plan generation algorithm, given an interior boundary and the room spec in Figure 7a. Here, we first divide the room into a "bedroom & bathroom" and a "kitchen & living room" zone. Then, within the "bedroom & bathroom" zone we place both the bedroom and bathroom, and within the "kitchen & living Room" zone, we place both the kitchen and living room.

kitchen and living room. The growth weight is used to approximate the size of each subdivision. By recursively subdividing the zones of each subtree, we satisfy the constraint that we can traverse between child nodes of the same parent in the room spec.

Finally, we scale the entire floor plan by $s \sim U(1.6, 2.2)$. Scaling the interior boundary to be larger provides more room for the agent to be able to navigate within the houses. Using a range of values also provides more variability on the size of the houses. We set the upper bound to 2.2 based on the empirical quality of the houses, where values above that often left too much empty space.

B.4 Connecting Rooms

Figure 11: An example of the 3 ways to connect different rooms, using either a doorway (left), door frame (middle), or open room connection (right).

Figure 11 shows the 3 types of ways adjacent rooms may be connected. Specifically, rooms may be connected using 3 different types of connections: doorways, door frames, or open room connections. We determine which rooms should have doors between them based on the constraints in the room spec. Amongst adjacent rooms that may have doors between them, subject to the constraints in the room spec, we randomly sample which rooms have doors. We also impose the constraint that neighboring rooms in the room spec may have at most 1 room connected to it.

To choose the type of connection, we consider the rooms we are connecting. Specifically, we only allow open room connections and door frame connections between kitchen and living room room types. We impose this constraint because it would be unrealistic for a room like a bathroom to be fully visible from another room. For connecting room types that do support open room connections or door frames, we annotate the probability of sampling a doorway, door frame, and open room connection. Between a kitchen and living room the probability is 0.375 for sampling both an open room connection and a door frame connection, and 0.25 for sampling a doorway connection.

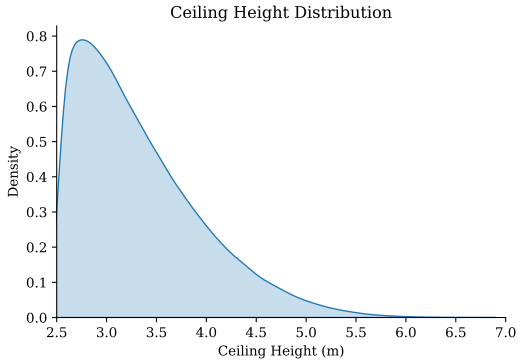


Figure 12: The distribution of the ceiling height of each house, in meters.

If a doorway or door frame is sampled, we filter to use a valid asset that is smaller than the wall connecting the rooms. For our generation, the minimum wall size is always greater than a single door size, but occasionally the filter might remove double doors from valid doors that can be sampled as they would be too big. The placement of the door is then uniformly sampled from anywhere along the wall. For doorways, the open direction is uniformly sampled. Finally, if the open state from any 2 doorways collides, we also use rejection sampling to potentially change the open direction and modify the placement of doorways.

Each house also has a permanently closed exterior door connecting to the outside. We prioritize placing this door in kitchen and living room room types, as it is unnatural to have to go through a bathroom or bedroom to go outside. However, in the case where the room spec does not include a kitchen or living room (*e.g.* if the room is a standalone bathroom), we randomly place a door to the outside in one of the remaining rooms.

B.5 Structure Materials

Wall materials. To choose the materials that make up the walls, we consider 2 families of wall materials: solid colors and texture-based materials. Our solid color materials consist of 40 unique colors of popular paint colors found in houses. We constrain ourselves to only using popular paint colors, so we do not randomize the walls to unrealistic colors such as bright green or yellow. For the texture-based materials, we annotate 122 different AI2-THOR materials to be suitable as wall materials. These include materials for brick textures, drywall textures, and tiling textures, amongst others.

Each wall in a room shares the same materials. For each room, we sample it if its materials are a solid color with $w_{solid} \sim \text{Bernoulli}(0.5)$. It is sometimes the case in real life that all rooms in a house share the same material (*e.g.* every room in an apartment is painted with white walls). We therefore also have a parameter $w_{same} \sim \text{Bernoulli}(0.35)$ that specifies if all rooms in the house will have the same material.

Ceiling material. The entire ceiling of the house is always assigned to a single wall material. If w_{same}, then the ceiling material is also set to the wall material. Otherwise, it is independently sampled with the same wall material sampling process.

Floor materials. We annotate 55 materials in AI2-THOR as floor materials. Most commonly, these materials are wood materials. For each room, we independently sample its floor material from the set of annotated floor materials. However, similar to wall materials, we independently sample $f_{same} \sim \text{Bernoulli}(0.15)$ that specifies if all rooms in the house will have the same material.

B.6 Ceiling Height

The ceiling height for the house, in meters, is sampled from $c_h \sim h_{\min} + (h_{\max} - h_{\min}) \cdot \text{Beta}(\alpha_h, \beta_h)$, where we set $h_{\min} = 2.5$, $h_{\max} = 7$, $\alpha_h = 1.25$, and $\beta_h = 5.5$. Figure 12 shows the ceiling height distribution that is formed. All rooms in the house have the same ceiling height.

The minimum and mean values were chosen based on the typical height of an American apartment, while β_h allows some of the train houses to have much larger ceilings.

B.7 Lighting

Lighting Placement. Each procedural house places two types of lights: a directional light and point lights. The directional light is analogous to the sun in the scene, where only 1 is placed in each scene. Light from point lights are analogous to the light emitted from lightbulbs. We place a point light in each room near the ceiling, centered at the centroid of the room's floor polygon. Using the centroid ensures that the light is always placed inside of the room, even for L-shaped rooms. Additionally, desk lamp and floor lamp objects have a point light associated with them.

Figure 13: Examples different skyboxes in a scene with a midday skybox (left), golden hour skybox (middle), and a blue hour skybox (right). Notice how the colors of the images differ and how the content outside of the window changes with the skybox.

Effects by the time of day. Skyboxes may appear at 3 different times of day: midday, golden hour, and blue hour. The time of day determines the intensity, hue, and direction of the ambient outdoor lighting. For each time of day, there exist multiple *skyboxes*, which dictate the lighting of the environment. Figure 13 shows examples of how the time of day visually affects the scene. At this time, there are 16 midday skyboxes, 5 golden hour skyboxes, and 1 blue hour skybox, based on full 360-degree photos taken in Seattle and San Francisco.

B.8 Object Placement

In this section, we discuss how objects are placed realistically in the house. We hypothesize reasonable object placement is necessary in order to train efficient agents. For instance, if a toilet could appear anywhere in the house, the agent would have a much harder search problem, leading to longer episodes, than if the toilet was always in the bathroom. Moreover, we do not want objects to appear in unnatural positions, such as a fridge facing the wall, as it would make it unnatural, and even unusable, for interaction.

Finally, we do not always want objects to spawn independently. For instance, we might want a table to be surrounded by chairs. We achieve dependant sampling by developing SAGs, which are described in the section that follows.

B.8.1 Assets

The ProcTHOR asset database consists of 1,633 interactive household assets across 108 object types (see Appendix A for more details). The majority of assets come from AI2-THOR. Windows, doors, and counter tops are built into the exterior of rooms in AI2-THOR, which prevents us from spawning them in as standalone assets. Thus, we have also hand-built 21 windows, 20 doors, and 33 counter tops.

Asset Annotations. Our assets include several annotations that help us place them realistically in a house. Figure 14 shows an example of the asset annotations used to place an arm chair. For an individual asset, we annotate its object type, computationally obtain its 3D bounding box, and partition assets of object types into training, validation, and testing splits. Then, we annotate how

each object type might be spawned into the house. Annotating the 108 object types, as opposed to annotating the 1,633 individual assets, allows us to scale up the number of unique assets dramatically. Moreover, it does not require any new annotation to add an asset that can be grouped with an existing object type.

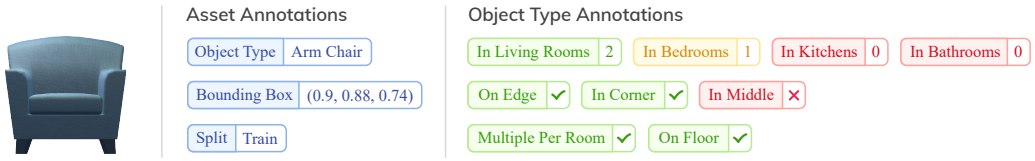


Figure 14: An example of the asset annotations used to place an arm chair asset. This particular instance is annotated with its object type, bounding box, and split. Annotations about how it is placed in the house are done at an object type level, applying to all instances of that type.

If instances of an object type cannot be placed independently on the floor, the rest of its annotations are not considered. For instance, we do not allow television object types to be placed alone on the floor, rather they are often placed on top of a television stand or mounted on the wall, which is discussed later in this section. Similarly, we also annotate small objects, like a fork, pen, and mug to not be placed independently on the floor. However, typical large object types, such as counter top, arm chair, or fridge object types can be placed independently on the floor.

Among the remaining object types, we annotate where and in which rooms the object type may appear. Each object type has a room weight, $r_w \in \{0, 1, 2, 3\}$, corresponding to how likely it is to appear in each room type. For each room type, a 0 indicates the object should never appear (e.g., a fridge in a bathroom); a 1 indicates the object may appear, but is unlikely; a 2 indicates that the object appears quite often; and a 3 indicates that the object nearly always appears (e.g., a bed in a bedroom). To determine where the object is placed, we annotate whether it may appear on the edge, in the corner, or in the middle of a room. For example, we annotate that a fridge can be placed on the edge or in the corner of the room, but not in the middle. We also annotate whether there can be multiple instances of an object type in a single room. Here, we annotate that multiple toilet object types cannot be in the same room, for instance.

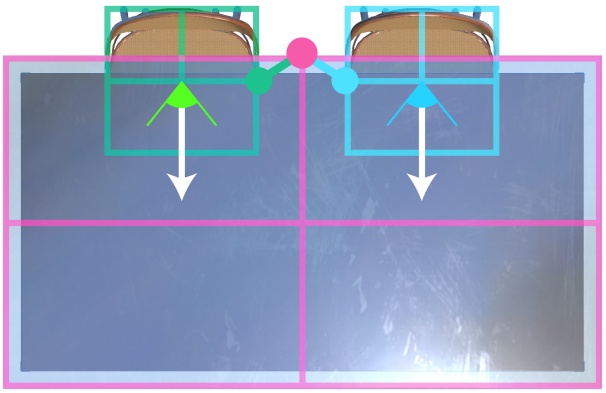
Asset Splits. If an object type has over 5 unique assets, then those assets are partitioned into train, validation, and testing splits. Specifically, approximately $2/3$ of the assets are assigned to the train split, and approximately $1/6$ of the assets are assigned to each of the validation and testing splits. For object types that have 5 or fewer unique assets, they may appear in any split. In general, the more visual diversity an object type has, the more instances of that object type exist. For instance, there are many chair objects, but there are much fewer CD, toilet, and fork objects. Appendix A shows the precise count of each object type.

B.8.2 Semantic Asset Groups (SAGs)

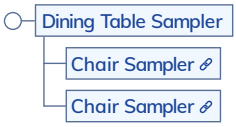
A *Semantic Asset Group* (SAG) provides a flexible and diverse way to encode which objects may appear near each other. The power of SAGs comes in their ability to support randomized asset and rotational sampling. SAGs can be created and exported in seconds with our user-friendly drag-and-drop web interface.

Figure 15 shows an example of how we might construct a SAG that has two chairs pushed into the side of a dining table. The SAG includes two chair samplers and a dining table sampler. Asset samplers contain a set of unique 3D modeled asset instances that may be sampled. When the SAG is instantiated, each asset sampler randomly chooses one of its instances. Asset samplers can also be linked, where multiple samplers sample the same asset instance each time. Here, linking may allow for multiple instances of the same chair to be placed at a dining table, instead of independently sampling a different chair for each sampler.

The ability to randomly sample assets to place in a SAG is incredibly expressive. For instance, consider a SAG with samplers for a TV stand, television, sofa, and arm chair. If each of these samplers can sample from just 30 different 3D modeled asset instances, then there are over 800k unique combinations of instances that can make be sampled from that SAG.



(a) An interface for viewing SAGs showing child asset anchoring and rotational randomness.

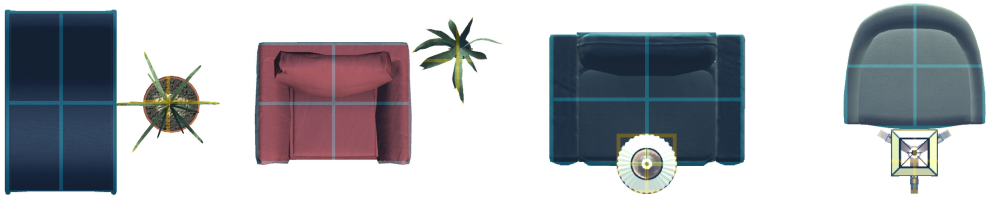


(b) Hierarchy

(c) Annotation

Figure 15: An example of a semantic asset group (SAG), where two chair samplers are parented to a dining table sampler. Both chairs are anchored to the top middle of the table.

Asset samplers define how assets are positioned relative to one another. SAGs are constructed by looking at instances of asset samplers from their top-down orthographic images, such as the one shown in Figure 15a. Here, both of the chair samplers are parented to the dining table sampler. Each child asset sampler is anchored to its parent asset sampler vertically in $\mathcal{V} = \{\text{TOP}, \text{CENTER}, \text{BOTTOM}\}$ and horizontally in $\mathcal{H} = \{\text{LEFT}, \text{CENTER}, \text{RIGHT}\}$. Each child asset sampler's pivot position can similarly be set vertically in $\mathcal{V}$ and horizontally in $\mathcal{H}$. For instance, in Figure 15a, both chair samplers are anchored to the parent vertically on TOP and horizontally in the CENTER. But, the chair sampler on the left's pivot position is vertically in the CENTER and horizontally on the RIGHT, whereas the chair sampler on the right's pivot position is vertically in the CENTER and horizontally on the LEFT. Figure 16 shows more examples of how a plant or floor lamp sampler may be positioned around an arm chair sampler. Each child asset sampler can then have an (x, y) offset, which is the distance from the parent sampler's anchor point to the child sampler's pivot position.



(a) CENTER RIGHT Anchor CENTER LEFT Pivot

(b) CENTER RIGHT Anchor BOTTOM LEFT Pivot

(c) BOTTOM CENTER Anchor CENTER CENTER Pivot

(d) BOTTOM CENTER Anchor TOP CENTER Pivot

Figure 16: Instantiations of a SAG that places a plant or floor lamp sampler $\mathcal{S}_c$ around a parented arm chair sampler $\mathcal{S}_p$ with anchor and pivot position annotations. Notice that the placement from $\mathcal{S}_c$ reacts to the size of the asset sampled from $\mathcal{S}_p$. None of the examples have any offset.

The motivation for the relative positioning of asset samplers is to prevent the meshes from clipping into each other. For instance, with the same SAG in Figure 15a, consider what would happen if the dining table sampler samples a table that is double the size of the current table. Instead of the chairs being stuck in a fixed global position, and effectively colliding with the new dining table, the chairs will reactively move back, and be re-positioned to remain slightly tucked under the larger table. Moreover, consider that the size of instances that are sampled from an asset sampler are often quite different. For instance, one table might be square-ish, while another is elongated. If we only used a CENTER CENTER pivot and an offset, one would not be able to reliably place asset samplers, containing differently sized objects, directly beside each other without it resulting in clipping.

While setting anchoring and pivot positions solves many mesh clipping issues, some cases may still arise. Figure 17 shows an example, where if our dining table sampler samples a short dining table, it

(a) ✗ Rejected (b) ✓ Accepted

Figure 17: Rejection sampling is used to make sure objects placed in SAGs do not collide. *Left:* the chair collides with the dining table, and hence it is rejected; *Right:* none of the objects in the instantiated SAG collide with each other, so the SAG is accepted as valid.

may clip into certain chairs. Such issues are rare in practice, but object clipping would lead to less realistic and interactive houses. To solve the clipping issue, we use rejection sampling to resample the assets of a SAG until none of the 3D meshes of the sampled assets are clipping.

In PROCTHOR-10K, we construct 18 SAGs, which can be instantiated with over 20 million unique combinations of assets. These include semantic asset groups for chairs around tables, pillows on top of beds, sofas and arm chairs looking at a television on top of a TV stand, faucets on top of sinks, and a desk with a chair, amongst others.

B.8.3 Floor Object Placement

We start object placement by first placing objects on the floor of the house. Objects are independently placed on a room-by-room basis, where we may first place objects in the bedroom and then place objects in the bathroom, without either affecting each other.

For each room, we filter the objects down into only using objects that have a room weight $r_w > 0$ in the given room type, and that have the annotation that they can be placed on the floor. Here, for instance, a chair object may have the annotation that it can be placed on the floor, but a knife object may not.

At this stage, we simplify rooms to just look at the top-down 2D bounding box that makes up the room in the floor plan. We also simplify objects to just look at its top-down 2D bounding box, of size (o_w, o_h). These simplifications make it easier to determine if an object will fit in the room, specifically in a particular rectangle.

Figure 18 illustrates the iterative process of placing objects in the scene. First, the polygon forming the area left to place an object is partitioned into rectangles. The rectangles come from drawing a horizontal and vertical grid line at all corner points of the open polygon. Here, we can easily obtain the largest rectangle remaining in the open room polygon. We sample $r_\ell \sim \text{Bernoulli}(0.8)$ to determine if the next object to be placed should be placed inside of the largest rectangle. Otherwise, we randomly choose amongst all possible rectangles, weighted by the area of each rectangle.

Once we have the rectangle (r_w, r_h) where the object should be placed in, we filter our objects to only those that would fit, both semantically and physically, in the rectangle. Semantically, we consider 3 scenarios: the rectangle being on the corner, edge, or middle of the room's polygon.

If any of the rectangle's corners is in a corner of the room, then we will place an object in that corner of the room. If multiple of the rectangle's corners are in a corner of the room, then we uniformly sample a corner amongst one of those corners.

Now, we will filter down objects and asset groups to only consider:

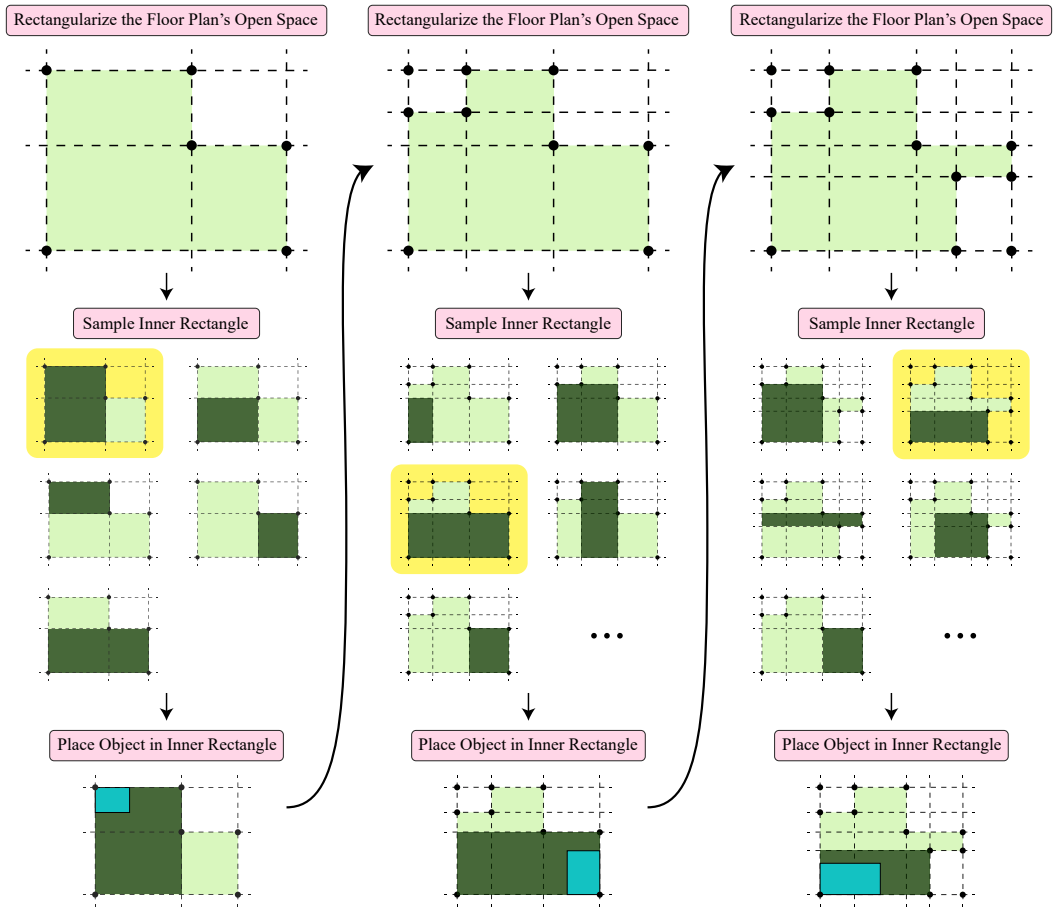
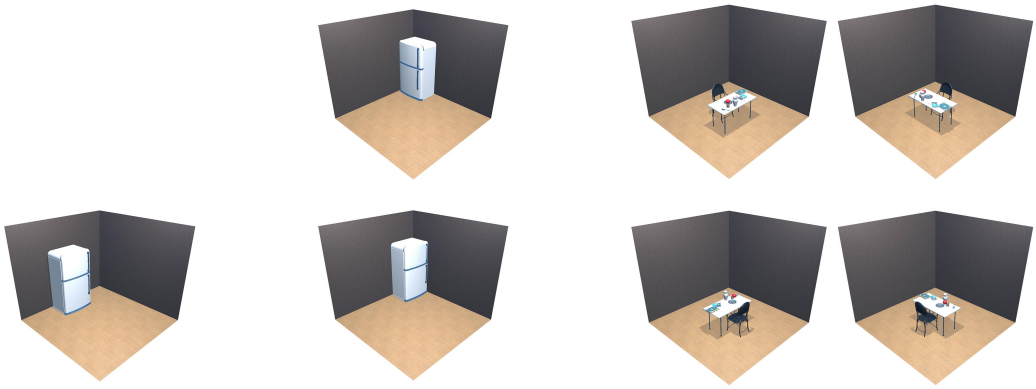


Figure 18: Diagram detailing how floor objects are placed in a room. First, we rectangularize the top-down view of the room's open floor plan by drawing horizontal and vertical dividers from each corner point. Then, we construct all possible rectangles that are formed within the dividers. We then sample one of those rectangles and place the object within that rectangle. The sampled object's top-down bounding box (with margin) is shown in blue. The bounding box is then subtracted from the open floor plan before repeating the process again.

1. Those that are annotated specifying that they can be placed in the corner of the room. For example, we might annotate a fridge to be placed in the corner of the room, but we might not annotate a SAG consisting of a dining table to be placed in the corner of the room.
2. The annotated split of the asset instance matches the current split of the generated house. See Appendix B.8.1 which talks about asset splits to create train/val/test homes.
3. The top-down bounding box of the object (with margin) must fit within the chosen rectangle. For a corner object, Figure 19b shows the 2 valid rotations that this object may take on. Specifically, the back of the object may be against either wall. Then, we filter down remaining objects to only use those where the object's bounding box fits within the rectangle's bounding box; that is, $(o_h + w_{pad} \leq r_w$ and $o_w + w_{pad} \leq r_h)$ or $(o_h + w_{pad} \leq r_h$ and $o_w + w_{pad} \leq r_w)$. If both conditions are valid, we uniformly choose one of the rotations of the object's bounding box.

   We add margin around objects to make sure it is always possible to navigate around them. Objects to be placed in the middle of the room have $m_{pad} = 0.35$ meters of margin on each side. Objects on the edge or corner of the room have $w_{pad} = 0.5$ meters of margin only in front of the object, which enables objects to be placed directly beside it.

We sample an object or asset group that satisfies all of the previous conditions. If there are no objects or asset groups that satisfy all conditions, we continue to the next iteration and remove the selected rectangle from consideration. We slightly prioritize placing asset groups over standalone assets when



(a) Edge Rotations (b) Corner Rotations (c) Middle Rotations

Figure 19: Valid rotations of objects when placed on the edge, corner, and middle of the room. Objects placed on the edge or corner of the room always have their backs to the wall. Objects in the middle of the scene can be rotated in any direction. By constraining rotations of objects, we ensure an object on the edge of the room, such as a fridge or drawer, can still be opened.

possible. Once we have chosen an object or asset group, the bounding box with margin is then anchored to the corner of the rectangle, and hence to the corner of the room. We then subtract the object's bounding box, with margin, from the open polygon representing the space remaining in the room before doing the same process again.

If the rectangle is along the edge, we sample $r_{edge} \sim \text{Bernoulli}(0.7)$ to determine if we should try to place an object on the edge of the rectangle, or if we should try and place it in the middle. If the rectangle is not along the edge or on the corner of the room, then we will always try to place an object in the middle of it. We use a similar filtering process, as the one described with edge rectangles, to filter down objects to those that only fit within the bounds of the rectangle. However, as depicted in Figure 19a and Figure 19c, edge objects can only have their backs to the wall, and middle objects can be rotated in any 90-degree rotation.

The iterative process of sampling a rectangle from the open polygon of the room, placing an object in that rectangle, and subtracting the bounding box formed by the object in the rectangle, continues on for r_i, where r_i is sampled from

$$r_i \sim \begin{cases} 1 & p = 1/200 \\ 4 & p = 2/200 \\ 5 & p = 4/200 \\ 6 & p = 20/200 \\ 7 & p = 173/200 \end{cases}. \tag{1}$$

Sampling r_i allows us to infrequently have rooms in the house where there are very few objects, which is sometimes the case in real-world homes. It should also be noted that there can be more than r_i objects on the floor of the scene if some objects in the scene are in SAGs.

By iteratively choosing the largest, or near largest, rectangle in the room's open polygon, placing an object in it, and subtracting the object's bounding box with margin from the open room polygon, we enable great coverage across the entirety of the room, and hence the entirety of the house.

B.8.4 Wall Object Placement

After placing objects on floors, we then place objects on walls. We currently place window, painting, and television objects on the walls. Figure 20 shows some examples. Window and television objects may appear in kitchen, living room, and bedroom room types. Paintings may appear in any room type.

Windows. Window objects are the first objects we place on the walls of the house. We only consider placing a window on walls that are connected to the outside of the house, such that we do not place a

Figure 20: Examples of objects placed on the wall of a house, including a window (left), painting (middle), and television (right).

window between two indoor rooms. For each kitchen, living room, and bedroom in the house, we sample

$$n_w \sim \begin{cases} 0 & p = 0.125 \\ 1 & p = 0.375 \\ 2 & p = 0.5 \end{cases} \tag{2}$$

maximum window objects to be placed.

For each wall in a given room, we look at the segment formed by each edge connecting 2 adjacent corners. If there is a floor object placed along that edge (or corner) of the wall, we subtract it from the segment. Here, the segment may break into different segments, where each segment is treated just like the original one. If the length of any segment is smaller than the minimum window size in the split, we remove the segment. We then use a uniform sample over the remaining segments, weighted by their lengths, to determine where to place the window. If no segments are longer than the smallest window, we move on to the next room in the house. A window smaller than the length of the segment is then uniformly placed somewhere along the sampled segment. The window is vertically centered along the wall between the floor and $w_{\max} = \min(3, c_h)$. All segments along the wall where the window was placed are removed from future sampling calls, and we continue this process n_w times.

Paintings. Painting objects are placed on the walls after window objects. They may be placed in any room. The maximum number of painting objects that are attempted to be placed in each room is sampled from

$$n_p \sim \begin{cases} 0 & p = 0.05 \\ 1 & p = 0.1 \\ 2 & p = 0.5 \\ 3 & p = 0.25 \\ 4 & p = 0.1 \end{cases}. \tag{3}$$

The placement of painting objects is similar to the placement of window objects. However, multiple painting objects may be placed along the same wall, so instead of removing the entire wall segment after an object is placed on it, we subtract the width of the painting from the segment. Moreover, we also allow painting objects to be placed above edge floor objects if the height of the edge object is less than 1.15 meters. Here, this allows for a painting to be above an object like a counter top, but not behind a taller object like a fridge.

The vertical position of each painting is sampled at $o_y \sim w_{\min} + (w_{\max} - w_{\min}) \cdot \text{Beta}(12, 12)$, where $w_{\min}$ is the maximum height of a floor object along the wall line. Here, we allow a painting to be placed above an object along the wall of the room, such as placing it above a counter top. Sampling from $\text{Beta}(12, 12)$ allows for some randomness in the sampling process while still having a large density near the center.

Televisions. Television wall objects may only be placed in living room, kitchen, and bedroom room types. Only 1 wall television may be placed in each room. From our annotations, television objects cannot be placed standalone on the floor. However, a television is often placed in a SAG, on top

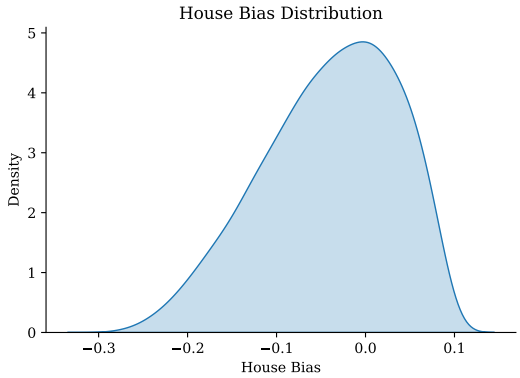


Figure 21: The house bias distribution b_{house} that offsets the probability of attempting to spawn an object in a receptacle.

of an object like a TV stand. So as to not place too many television objects in the same room, we only filter by rooms that do not have a television object already in them. Amongst the remaining rooms, if the room type is a living room, we sample Bernoulli(0.8) if we should try placing a wall television in the room. For kitchen and bedroom room types, we sample from Bernoulli(0.25) and Bernoulli(0.4), respectively. We only consider television objects that could be mounted to a wall (*i.e.* they do not have a base that is sticking out of the object). Television wall objects sample from the same vertical position distribution as painting objects, and follow the same placement on the walls as painting objects.

B.8.5 Surface Object Placement

After placing objects on the floor and wall of the house, we focus on placing objects on the surface of the floor objects just placed. For example, we may place objects like a coffee machine, plate, or knife on of a receptacle like a counter top.

For each receptacle object, we approximate the probability that each object type appears on its surface. We use the hand-modeled AI2-iTHOR or RoboTHOR rooms to obtain these approximations. Here, we compute the total number of times each object type is on the receptacle type and divide it by the total number of times the receptacle type appears across the scenes.

For each receptacle placed on the floor, we look at the probability of each object type p_{spawn} that it has been placed on that receptacle. We then iterate over the object types that may be on the receptacle. For each object type, we try spawning it on the receptacle if Bernoulli$(p_{spawn} + b_{house} + b_{receptacle} + b_{object})$, where

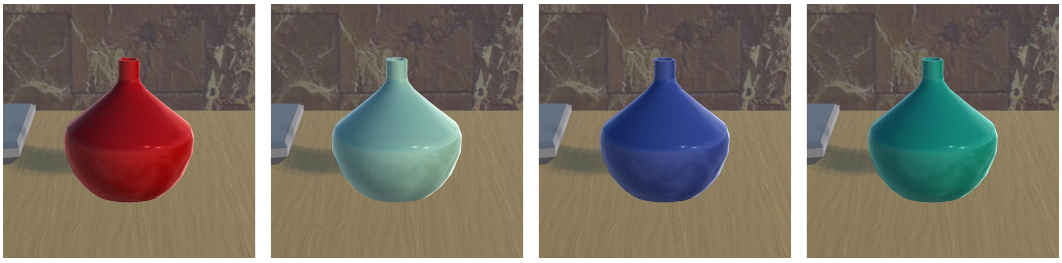
- b_{house} denotes the additional bias of how likely objects are to be spawned on receptacles in this particular house. Each house samples

$$b_{house} \sim (b_{house\text{-}max} - b_{house\text{-}min}) \cdot \text{Beta}(3.5, 1.9) + b_{house\text{-}min}, \tag{4}$$

  where $b_{house\text{-}min} = -0.3$ and $b_{house\text{-}max} = 0.1$. Figure 21 shows the distribution that b_{house} forms. Using a house bias allows for some houses to be much cleaner or dirtier than others, whereas cleaner houses would have more objects put away that are not on receptacles.

- $b_{receptacle}$ denotes the additional bias of how likely an object is to be spawned on a receptacle. The default receptacle bias is 0.2, which is only overwritten by shelving unit (0.4 bias), counter top (0.2 bias), arm chair (0 bias), and chair (0 bias). Receptacle biases were manually set based on the empirical quality of the houses.

- b_{object} denotes the additional bias of how likely a particular object is to spawn in the scene. By default, b_{object} is set to 0, and overwritten by house plant (0.25 bias), basketball (0.2 bias), spray bottle (0.2 bias), pot (0.1 bias), pan (0.1 bias), bowl (0.05 bias), and baseball bat (0.1 bias). Object biases were also manually set based on the empirical quality of the houses to ensure more target objects appear in each of the procedurally generated houses.



(a) Examples of color randomization for a vase object. The original color is shown on the left. Notice that the vase still looks realistic with many possible colors.

(b) Examples of material randomization in ProcTHOR. Notice that only the objects randomize in materials, where the walls, floor, and ceiling remain the same.

Figure 22: Examples of color randomization and material randomization in ProcTHOR.

Note that $p_{spawn} + b_{house} + b_{receptacle} + b_{object}$ may be greater than 1, in which case we will always try to spawn the object on the receptacle, or less than 0, where we will never try to spawn the object on the receptacle.

To attempt to spawn an object of a given type on a receptacle, we will sample an instance of that object type and randomly try $n_{pa} = 5$ poses of the object to try and fit the object instance on the receptacle. If the object instance fits and does not collide with another object, we keep it there. Otherwise, we try another pose of the object on the receptacle until we reach n_{pa} attempted poses. If none of the attempted poses work, we continue on to the next object type that may be on the receptacle.

If the first object of a given type is placed successfully on a receptacle, we attempt to place $n_{or} \sim \min(s_{\max}, \text{Geom}(p_{spawn}) - 1) - 1$ more objects of that type given type on the receptacle. Here, $s_{\max}$ is set to 3, representing the maximum number of objects of a type that may be on a receptacle. We ignore the biases to not have too many objects of a given type on the same receptacle.

B.9 Material and Color Randomization

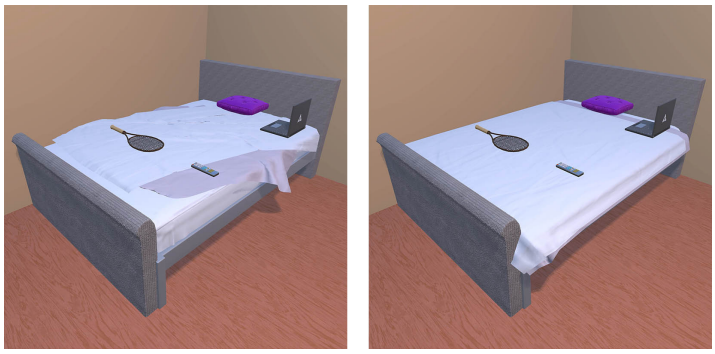
Several object types may have their color randomized to a randomly sampled RGB value. Specifically, for each vase, statue, or bottle in the scene, we independently sample from $r_c \sim \text{Bernoulli}(0.8)$ to determine if we should randomize the object's color. These objects were chosen because they all still looked natural as any solid color. Figure 22a shows some examples of randomizing the color of a vase.

For each training episode, we sample from $r_m \sim \text{Bernoulli}(0.8)$ to determine if we should randomize the default object materials in the scene. Wall, ceiling, and floor materials are left untouched to preserve w_{solid} and w_{same} sampling parameters. Materials are only randomized within semantically similar classes, which ensures objects still look and behave like the class they represent. For instance, an apple will not swap materials with an orange. Figure 22b shows some examples of randomizing the materials in the scene.

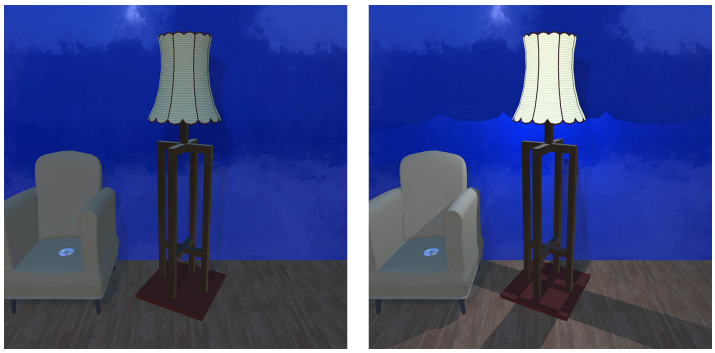
B.10 Object States

We randomize object states to expose the agent to more diverse objects during training. For instance, instead of always having an open laptop or a clean bed, we randomize the openness of each laptop and

(a) Openness state randomness example with a laptop.



(b) Clean state randomness example with a bed.



(c) On or off state randomness with a floor lamp.

Figure 23: Examples of object state randomness.

if each bed is clean or dirty. Figure 23 shows some examples. Our current set of state randomizations include:

- **Toggling objects.** Floor lamp and desk lamp object types have their state toggled on or off.
- **Cleaning or dirtying objects.** Bed object types may appear as either clean or dirty.
- **Opening or closing objects.** Box and laptop object types may

toggling objects on or off (for floor lamp and desk lamp object types), setting objects to clean or dirty (for bed object types), and openness randomizations (for box and laptop object types).

B.11 Validator

Once a house is generated, we use a validator to make sure that the agent can successfully navigate to each room in the house, without modifying the scene through interaction (*e.g.* moving an object out of the way). Specifically, we first make sure the agent can teleport to a location inside the house. Then, from that position, we perform a BFS over neighboring positions on a 0.25×0.25 meter grid to obtain all reachable positions from the agent's current position. The validator checks to make sure that every room in the house has at least 5 reachable positions on the grid. If the validator fails, we resample a new house using the same room spec, so as to not change the distribution of room specs that we sample from.

B.12 Related Works

In this work, our goal is to generate diverse and semantically plausible houses. We also aimed to make it easily extendable in the future, adapting to new object types or synthesizing new room types. To this end, we tried to use the best approaches or build on existing works that are insufficient for our use case. Given the modular nature of our house generation process, if a better algorithm exists at any stage of the pipeline, we can easily update the generation process with that algorithm to generate better houses.

Floorplan Generation. Floorplan generation involves taking a set of rooms to place in a house and an interior boundary (i.e., a top-down outline of the home) and partitioning the interior boundary into rooms. Floorplan generation is a longstanding problem, with many works generating floorplans with procedural generation [61, 4, 88, 30, 65] and deep learning [73, 74, 42, 103]. Our Floorplan generation algorithm is based on [61, 65], which generates a floorplan from a room specification, connectivity constraints between rooms, and an interior boundary. [88] is similar to [61], except that it tries to learn the approximate size of room types from data rather than manually specifying the relative sizes of each room. [42] proposes a similar approach that tries to generate a floorplan based on room preferences, room connectivity constraints, and an interior boundary, but it trains a network on RPLAN [103] to solve these constraints. [73, 74, 103] train a network to generate floorplans, but it does not support inputting any preferences about the number of rooms or the types of rooms in the house. However, in contrast to [61], such work cannot generate arbitrary floorplans that are out of the training distribution, which is problematic if one wanted to generate new types of rooms, such as garages and stairways connecting to another floor [30], or generate massive multi-family floorplans. We also sample an interior boundary for each new house to generate more diverse floorplans.

Object Placement. Object placement involves selecting which objects from a given object database should appear in the house and arranging those objects in a plausible configuration within the rooms of a home (*e.g.* chairs near tables, paintings on walls, toilets in bathrooms). We built a 3-stage pipeline for placing objects, which (1) places objects on floors, (2) places objects on walls, and (3) places objects on the surface of other objects. Our approach requires specifying remarkably few constraints about how objects are placed in scenes, making it easily extendable to add new objects to our object database and generate new room types.

Many works studying object placement [67, 37, 109, 107, 47, 99, 28, 10, 11, 41, 64] relied on procedural generation. [67, 107, 47, 99] take a given set of objects and the outline of the room but iteratively optimize over several functions to try and minimize the cost function. The cost function determines how realistic the room is with respect to quantities such as how navigable it is and how far an object is from a wall. [109] uses a similar object placement algorithm to ours based on hierarchical relationships between objects. It tries to learn these relationships from 3D-Front [31], whereas we

specify constraints and SAGs for objects, such as which can be placed on walls and which objects may appear near each other. In [28], the authors take examples of object arrangements and generate similar ones using probabilistic models trained on 130 scenes. [37] introduces the idea of anchoring objects in parent-child relationships. For instance, objects may be anchored to a wall or on top of a surface such as a table. [10, 11, 41, 64, 8] take in text descriptions or graphs [62] of a furniture arrangement as input and attempt to place objects based on that. However, this work requires manually prompting the model for each new room one wants to generate, so it similarly does not scale well.

Some recent works have proposed using deep learning to place objects on the floors of rooms [96, 97, 79, 85, 110, 13, 31, 59, 78]. The main factors limiting our use of such models are that they: (1) cannot be easily adapted to place novel objects and room types and (2) the lack of high-quality training data of objects placed in 3D scenes. For training data, [96, 97, 85, 110, 59] uses SUNCG [92], a dataset that has been taken down due to legal issues, and [79, 13, 31, 78] uses 3D-Front. However, these approaches do not work with novel objects outside their training dataset and cannot generate novel room types that are not seen during training. Thus, it is impractical to use such approaches in ProcTHOR out of the box, as we have differing object databases. It is also impractical (and undesirable) to reproduce such approaches with our object database since we do not have large amounts of training data specifying examples of how our objects are placed in scenes. Here, even if we had such annotations, it would not allow us to add new objects to our object database in the future, as it would require manually collecting many new examples of where each is placed in scenes to train such models properly. Finally, note that kitchens and bathrooms are not diversely furnished in 3D-Front, so trying to learn object placement in such rooms is impractical.

The approach we use to place objects on surfaces is similar to that of [11, 37], where we calculate the co-object occurrence prior to determine which objects to place on a surface. For example, when looking at which objects to place on a dining table, then the co-object occurrence of a plate is much higher than that of a baseball bat. [48] proposes a fascinating approach to learning co-object occurrences using LLMs [22, 6, 108, 82]. It computes the probability of prompts such as "plate on table" or "baseball bat on table" to compute the relative probabilities of such pairwise combinations. In [28], they propose surface object placement by training probabilistic models that learn to cluster similar objects together as a way to scale much better to new object types. For example, they might input a cluttered desk with a chair, generating many new arrangements of a cluttered desk with new objects on the surface. [98, 29] have done randomizations of objects on surfaces without any priors about which objects should appear on a given surface.

We use semantic asset groups (SAGs) to place co-occurring objects next to each other. We define SAGs in an interactive web environment from top-down images of groups of objects. SAGs are most similar to object arrangements generated from Sketch2Scene [105], which takes an artistic sketch of a scene as input and generates plausible object configurations from that sketch. However, creating the sketches can be incredibly time-consuming, and sampling from them results in a leaky abstraction. In [107, 47, 99], the authors try to place select objects near each other by optimizing a pairwise distance constraint. Here, the cost function is set to minimize the distance between objects, such as chairs and a table, until they are sufficiently close. The relationships between the objects are manually defined at the object type level. SAGs are also similarly related to the idea of hyper relations in [37]. Instead of using positional anchoring, it uses density-based clustering to attempt to sample how objects are anchored around a parent object [86]. In addition, instead of manually defining the hyper-relations, they attempt to extract them from 3D-Front.

B.13 Limitations and Future Work

ProcTHOR-10K only generates 1-floor houses. We plan to support multi-floor houses in ProcTHOR-v2.0. This will allow us to capture a wider range of houses and provide better fine-tuning results. Additionally, we plan to scale up our asset databases by leveraging many open-source 3D asset databases, such as ABO [19], PartNet [69], ShapeNet [9], Google Scanned Objects [23], 3D-Future [32], and CO3D [84], among others.

ProcTHOR opens up many avenues of future research in scene synthesis targeted at training embodied agents. Along these lines, better leveraging real-world data as a prior, similar to what is done in Meta-Sim [49], is a promising direction. Similarly, using curriculum learning [76, 72] to train agents in environments that progressively get harder [1] may help train better and faster agents.

C PROCTHOR Datasheet

Motivation	
For what purpose was the dataset created?	The dataset was created to enable the training of simulated embodied agents in substantially more diverse environments.
Who created and funded the dataset?	This work was created and funded by the PRIOR team at Allen Institute for AI. See the contributions section for specific details.
Composition	
What do the instances that comprise the dataset represent?	Each house is specified as a JSON file, which specifies how to populate a 3D Unity scene in AI2-THOR.
How many instances are there in total (of each type, if appropriate)?	There are 10K houses released in the dataset, along with the code to sample substantially more. Section 4 shows the distribution of houses in PROCTHOR-10K.
Does the dataset contain all possible instances or is it a sample (not necessarily random) of instances from a larger set?	We make 10K houses available, but more houses can easily be sampled with the procedural generation scripts.
What data does each instance consist of?	Each house is specified as a JSON file, which precisely describes how our AI2-THOR build should create the house. The procedurally generated JSON files are typically several thousand lines long.
Is there a label or target associated with each instance?	No.
Is any information missing from individual instances?	No.
Are relationships between individual instances made explicit (e.g., users' movie ratings, social network links)?	Each house is generated independently, meaning there are no relationships between the houses.
Are there recommended data splits?	Yes. See Appendix B.8.1.
Are there any errors, sources of noise, or redundancies in the dataset?	No.
Is the dataset self-contained, or does it link to or otherwise rely on external resources (e.g., websites, tweets, other datasets)?	The dataset is self-contained.
Does the dataset contain data that might be considered confidential?	No.
Does the dataset contain data that, if viewed directly, might be offensive, insulting, threatening, or might otherwise cause anxiety?	No.
Collection Process	

How was the data associated with each instance acquired?	Each house was procedurally generated. See Appendix A.
If the dataset is a sample from a larger set, what was the sampling strategy?	The dataset consists of 1 million houses sampled from the procedural generation scripts.
Who was involved in the data collection process?	The authors were the only people involved in constructing the dataset.
Over what timeframe was the data collected?	Data was collected between the end of 2021 and the beginning of 2022.
Were any ethical review processes conducted?	No.
Preprocessing/Cleaning/Labeling	
Was any preprocessing/cleaning/labeling of the data done?	Section B.8 describes the labeling that was done to make the assets spawn in realistic places. We have also gone through every asset in the asset database to make sure the pivots for each asset are facing a consistent direction.
Was the "raw" data saved in addition to the preprocessed/cleaned/labeled data?	There is no raw data associated with the house JSON files.
Is the software that was used to preprocess/clean/label the data available?	The code to generate the houses is made available.
Uses	
Has the dataset been used for any tasks already?	Yes. See the Experiments section of the paper.

What (other) tasks could the dataset be used for?	The houses can be used in a wide variety of interactive tasks in embodied AI and computer vision. Any task that can be performed in AI2-THOR can be performed in ProcTHOR. For instance, in embodied AI, the houses may be used for navigation [51, 80, 101, 112, 100, 106, 63, 111], multi-agent interaction [44, 45, 1], rearrangement and interaction [98, 33, 35, 16, 93], manipulation [25, 75, 24, 104], Sim2Real transfer [20, 46, 58], embodied vision-and-language [91, 77, 43, 57, 38, 50], audio-visual navigation [15, 34, 14], and virtual reality interaction [102, 70, 40], among others. In the broader field of computer vision, the dataset may be used to study object detection [56]; NeRFs [68, 95, 39, 60]; segmentation, depth, and optimal flow estimation [27, 39]; generative modeling [52, 55, 54]; occlusion reasoning [26]; and pose estimation [12], among others. Our framework for loading in procedurally generated houses from a JSON spec also enables the study of scene clutter generation, building more realistic procedurally generated homes, and the development of synthetically generated spaces to train embodied agents in factories [71], offices, grocery stores [66], and full procedurally generated cities.
Is there anything about the composition of the dataset or the way it was collected and preprocessed/cleaned/labeled that might impact future uses?	No.
Are there tasks for which the dataset should not be used?	Our dataset may be used for both commercial and non-commercial purposes.

Distribution

Will the dataset be distributed to third parties outside of the entity on behalf of which the dataset was created?	Yes. We plan to make the entirety of the work open-source, including the code used to generate and load houses, the initial static dataset of 1 million procedurally generated house JSON files, and the asset and material databases.
How will the dataset be distributed?	The static house JSON files will be distributed with the PRIOR Python package [21]. The code, asset, and material databases will be distributed on GitHub.
Will the dataset be distributed under a copyright or other intellectual property (IP) license, and/or under applicable terms of use (ToU)?	The house dataset, 3D asset database, and generation code will be released under the Apache 2.0 license.
Have any third parties imposed IP-based or other restrictions on the data associated with the instances?	No.
Do any export controls or other regulatory restrictions apply to the dataset or to individual instances?	No.

Maintenance

Who will be supporting/hosting/maintaining the dataset?	The authors will be providing support, hosting, and maintaining the dataset.
How can the owner/curator/manager of the dataset be contacted?	For inquiries, email <mattd@allenai.org>.
Is there an erratum?	We will use GitHub issues to track issues with the dataset.
Will the dataset be updated?	We expect to continue adding support for new features to continue to make procedurally generated houses even more diverse and realistic. We also intend to support new tasks in the future.
If the dataset relates to people, are there applicable limits on the retention of the data associated with the instances (e.g., were the individuals in question told that their data would be retained for a fixed period of time and then deleted)?	The dataset does not relate to people.
Will older versions of the dataset continue to be supported/hosted/maintained?	Yes. Revision history will be available for older versions of the dataset.
If others want to extend/augment/build on/contribute to the dataset, is there a mechanism for them to do so?	Yes. The work will be open-sourced and we intend to provide support to help others use and build upon the dataset.

Table 1: A datasheet [36] for PROCTHOR and PROCTHOR-10K.

D ARCHITECTHOR

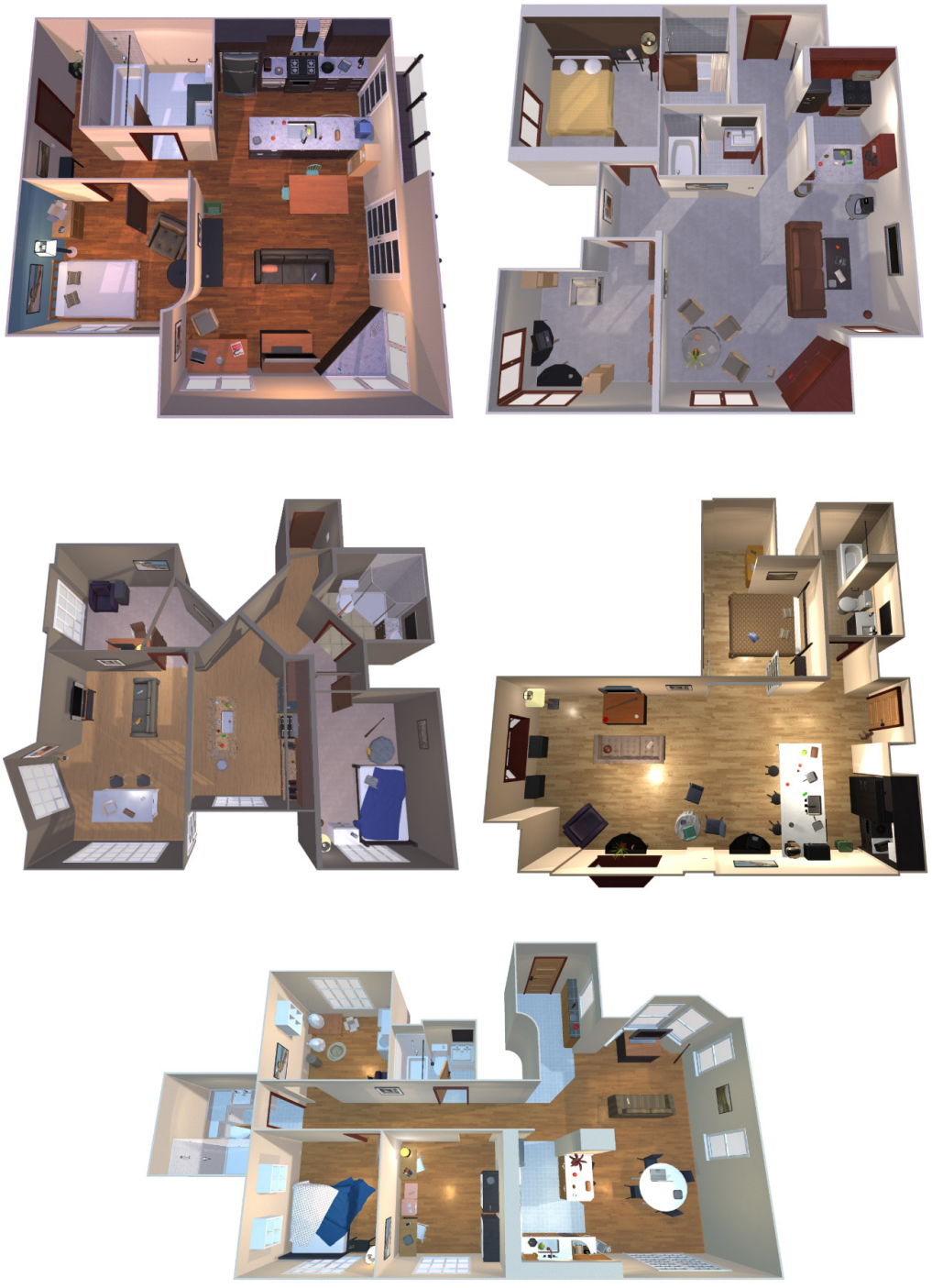


Figure 24: Top-down images of the 5 custom-built interactive validation houses in ARCHITECTHOR. The goal of these houses is to evaluate interactive agents in more realistic and larger home environments.

D.1 Datasheet

Motivation	
For what purpose was the dataset created?	ARCHITECTHOR was created to enable the evaluation of embodied agents in large, realistic, and interactive household environments.
Who created and funded the dataset?	This work was created and funded by the PRIOR team at Allen Institute for AI. See the contributions section for specific details.
Composition	
What do the instances that comprise the dataset represent?	Instances of the dataset comprise interactive 3D houses that were built in Unity and can be used with our custom build of the AI2-THOR API.
How many instances are there in total (of each type, if appropriate)?	There are 10 total houses, comprising 5 validation houses and 5 testing houses.
Does the dataset contain all possible instances or is it a sample (not necessarily random) of instances from a larger set?	The dataset is self-contained.
What data does each instance consist of?	Each instance of a house is a Unity scene, which includes data such as the placement of objects, lighting, and texturing.
Is there a label or target associated with each instance?	No.
Is any information missing from individual instances?	No.
Are relationships between individual instances made explicit (e.g., users' movie ratings, social network links)?	Each house was independently created.
Are there recommended data splits?	Yes. The houses themselves are partitioned as 5 validation houses and 5 testing houses. The assets placed in the house follow the same train/val/test splits used in PROCTHOR-10K.
Are there any errors, sources of noise, or redundancies in the dataset?	No.
Is the dataset self-contained, or does it link to or otherwise rely on external resources (e.g., websites, tweets, other datasets)?	The dataset is self-contained.
Does the dataset contain data that might be considered confidential?	No.
Does the dataset contain data that, if viewed directly, might be offensive, insulting, threatening, or might otherwise cause anxiety?	No.
Collection Process	

How was the data associated with each instance acquired?	Each house was professionally hand-modeled by 3D artists. Most objects placed in the hosues come from the PROCTHOR asset database. However, countertops, showers, and many cabinets were custom built.
If the dataset is a sample from a larger set, what was the sampling strategy?	The dataset consists of 1 million houses sampled from the procedural generation scripts.
Over what timeframe was the data collected?	The houses were built towards the beginning of 2022.
Were any ethical review processes conducted?	No.

Preprocessing/Cleaning/Labeling

Was any preprocessing/cleaning/labeling of the data done?	No.
Was the "raw" data saved in addition to the preprocessed/cleaned/labeled data?	There is no raw data associated with the ARCHITECTHOR houses.
Is the software that was used to preprocess/clean/label the data available?	Yes. We will open-source the ARCHITECTHOR houses and they can be opened and viewed in Unity.

Uses

Has the dataset been used for any tasks already?	Yes. Please see the Experiments section of the paper.
What (other) tasks could the dataset be used for?	The tasks can be used for any type of navigation and interaction tasks in embodied AI. The houses are built into our build of AI2-THOR, meaning ARCHITECTHOR can work with any task that can be performed in AI2-THOR. We especially think ARCHITECTHOR will be useful as an evaluation suite for evaluating different sets of PROCTHOR tasks and evaluating agents trained on different sets of procedurally generated houses.
Is there anything about the composition of the dataset or the way it was collected and preprocessed/cleaned/labeled that might impact future uses?	No.
Are there tasks for which the dataset should not be used?	Our dataset may be used for both commercial and non-commercial purposes.

Distribution

Will the dataset be distributed to third parties outside of the entity on behalf of which the dataset was created?	Yes. All houses in ARCHITECTHOR will be released to the open-source community and available through our build of the AI2-THOR Python API.
How will the dataset be distributed?	The houses will be distributed on GitHub and available to open as Unity scenes.

Will the dataset be distributed under a copyright or other intellectual property (IP) license, and/or under applicable terms of use (ToU)?	ARCHITECTHOR will be released under the Apache 2.0 license.
Have any third parties imposed IP-based or other restrictions on the data associated with the instances?	No.
Do any export controls or other regulatory restrictions apply to the dataset or to individual instances?	No.
Maintenance	
Who will be supporting/hosting/maintaining the dataset?	The authors will be providing support, hosting, and maintaining the dataset.
Is there an erratum?	We will use GitHub issues to track issues with the dataset once it is published.
Will the dataset be updated?	ARCHITECTHOR is currently in maintenance mode and we do not expect it to update much from its current state. However, we plan to actively support future AI2-THOR functionalities in ARCHITECTHOR, such as support for new robots, more advanced interaction capabilities, and bug fixes.
If the dataset relates to people, are there applicable limits on the retention of the data associated with the instances (e.g., were the individuals in question told that their data would be retained for a fixed period of time and then deleted)?	The dataset does not relate to people.
Will older versions of the dataset continue to be supported/hosted/maintained?	Yes. Revision history will be available in the GitHub repository.
If others want to extend/augment/build on/contribute to the dataset, is there a mechanism for them to do so?	Yes. The work will be open-sourced and we intend to provide support to help others use and build upon the dataset.

Table 2: A datasheet [36] for the artist-designed ARCHITECTHOR houses.

D.2 Analysis

ARCHITECTHOR consists of 10 remarkably high-quality large interactive 3D houses. Figure 24 shows top-down images of the 5 validation houses. Figure 25 shows some examples of images taken inside of 2 kitchens and a bedroom from ARCHITECTHOR validation.

ARCHITECTHOR was built to be much larger than AI2-iTHOR and RoboTHOR. Figure 26 shows the size comparisons between comparable hand-built scene datasets in AI2-iTHOR and RoboTHOR, measured in navigable area. Notice that the navigable area in ARCHITECTHOR is substantially larger than in those. The figure also shows the navigable areas in PROCTHOR-10K span the spectrum of navigable areas between AI2-iTHOR, RoboTHOR, and ARCHITECTHOR.

In total, the creation of the 10 houses in ARCHITECTHOR took approximately 320 hours of cumulative work by professional 3D artists. Figure 27 shows the time breakdown of which parts of the process took the longest. In particular, the creation of custom assets for the kitchen, such as modeling each of the countertops and cabinets, took the longest amount of time, followed by modeling the 3D structure of house.

Figure 25: Examples of images inside of 2 hand-modeled kitchens and 1 hand-modeled bathroom from ARCHITECTHOR validation.

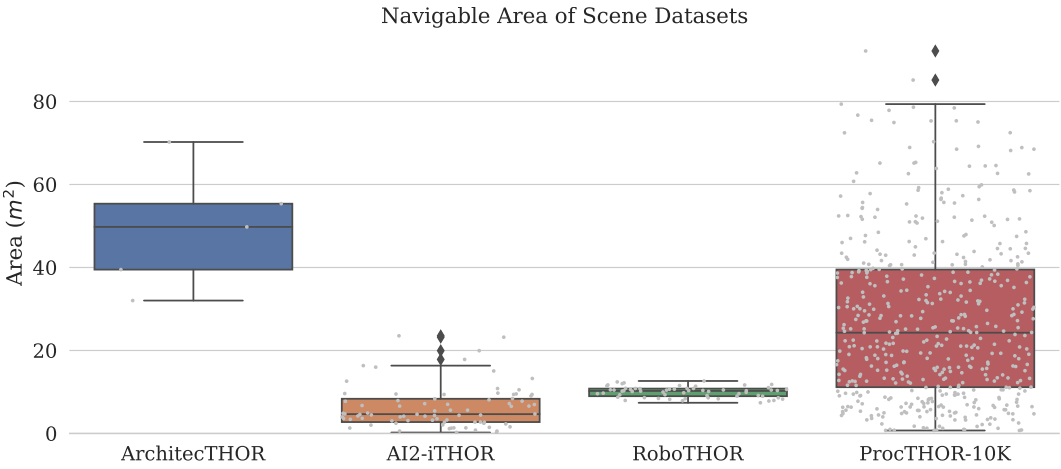


Figure 26: Box plots of the navigable areas for ARCHITECTHOR compared to AI2-iTHOR, RoboTHOR, and PROCTHOR-10K. Validation scenes were used to calculate the data for ARCHITECTHOR, and training scenes were used to calculate the data for AI2-iTHOR, RoboTHOR, and PROCTHOR-10K.

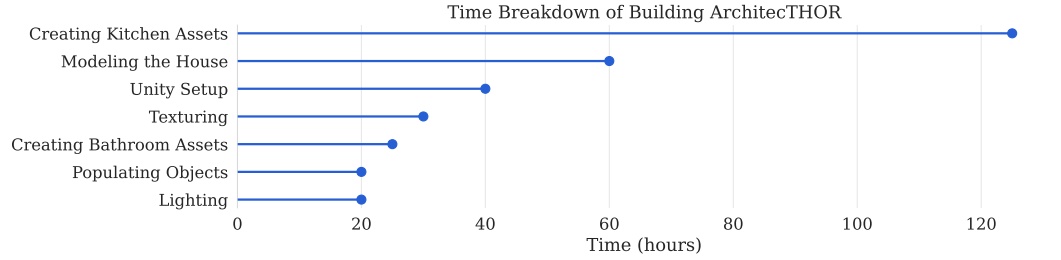


Figure 27: Cumulative time breakdown of the development of ARCHITECTHOR across 3D artists.

E Input Modalities

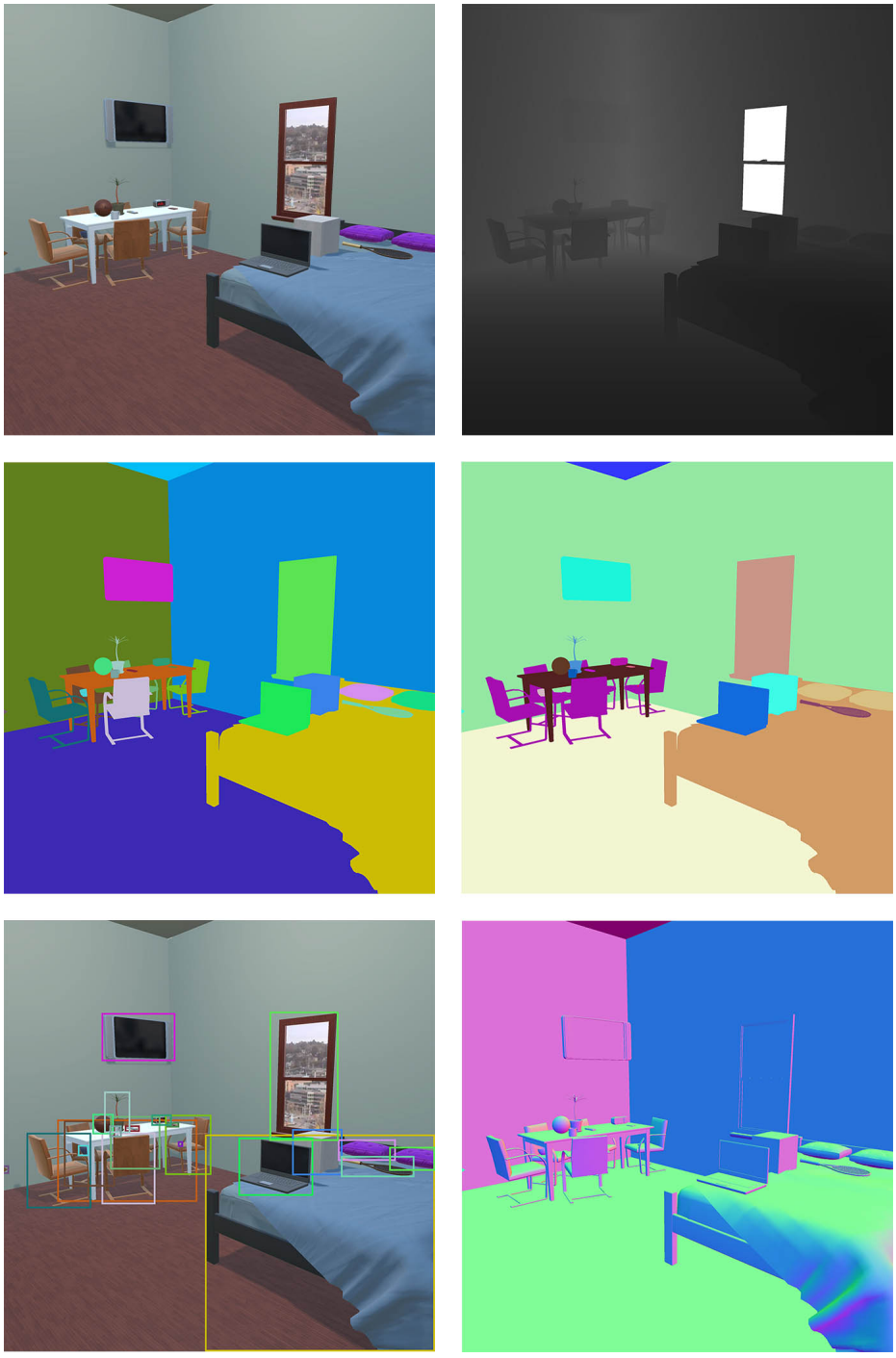


Figure 28: Examples of image-based modalities available in ProcTHOR include RGB (top left), depth (top right), instance segmentation (middle left), semantic segmentation (middle right), bounding box annotations (bottom left), and surface normals (bottom right). More image modalities can be added by modifying the Unity backend.

F Experiment details

This section discusses the training details used for our experiments. We discuss baselines, PROCTHOR pre-training, and environment-specific fine-tuning details for the tasks of ObjectNav, ArmPointNav, and rearrangement.

F.1 ObjectNav experiments

For ObjectNav experiments, agents are given a target object type (*e.g.* a bed) and are tasked with finding a path in the environment that navigates to that target object type. The task setup matches what is commonly used in embodied AI [20, 5, 51, 83], although we only utilize forward-facing egocentric RGB images at each time step. All ObjectNav experiments are trained with a simulated LoCoBot (Low Cost Robot) agent [7]. The task and training details are described below.

Evaluation. Following [3], an ObjectNav task is considered successful if all of the following conditions are met:

1. The agent terminates the episode by issuing the DONE action.
2. The target object type is within a distance of 1 meter from the agent's camera.
3. The object is visible in the final frame from the agent's camera. For instance, if (1) and (2) are satisfied, and the agent is looking in the direction of the object, but the target object is occluded behind a wall, then the task is unsuccessful. Similarly, if the target object type is located in the opposite direction of where the agent is looking, then the task will be unsuccessful.

We also use SPL to evaluate the efficiency of the agent's trajectory to the target object. SPL is defined and discussed in [3, 5]. A house may have multiple instances of objects for a given type that the agent can successfully reach. For instance, a house may have multiple bedrooms, where each bedroom includes a bed. Here, if the agent navigates to any of the beds, the episode is successful. To calculate SPL in these scenarios, the shortest path length for the task is the minimum shortest path length from the starting position of the agent to any of the reachable target objects of the given type, regardless of which instance the agent navigates towards.

Actions. For each of the trained models, we use a discrete action space consisting of 6 actions, which is shown in Table 3. Following common practice [20, 46], we use stochastic actuation to better simulate noise in the real world.

Action	Description
MOVEAHEAD	Attempts to move the agent forward by $\delta_m \sim \mathcal{N}(\mu = 0.25, \sigma = 0.01)$ meters from its current facing direction. If moving the agent forward by δ_m meters results in a collision in the scene (*e.g.* there is a wall directly in-front of the agent within δ_m meters), the action fails and the agent's position remains unchanged.
ROTATERIGHT ROTATELEFT	Rotates the agent rightwards or leftwards from its current forward facing direction by $\delta_r \sim \mathcal{N}(\mu = 30, \sigma = 0.5)$ degrees.
LOOKUP LOOKDOWN	Tilts the agent's camera up or down by 30 degrees.
DONE	A signal from the agent to terminate the episode and evaluate the trajectory from its current state. Discussed in [3].

Table 3: The action space for ObjectNav experiments.

Model. We use the relatively simple EmbCLIP [51] training setup for training all ObjectNav experiments. Table 4 shows the hyperparameters used during training, which are adapted from [51].

Except for the "ProcTHOR+Large" model trained for HM3D (described below), we otherwise use the same model architecture across ObjectNav experiments. Namely, at each time step, the agent receives a $3 \times 224 \times 224$ egocentric RGB image from its camera. The image is processed with a frozen RN50 CLIP-ResNet visual encoder [81] to produce a $2048 \times 7 \times 7$ visual embedding, $\mathbf{V}_t$. The embedding is compressed through a 2-layer CNN (going from 2048 to 128 to 32 channels) with 1×1 convolutions [94] to obtain a $32 \times 7 \times 7$ tensor, $\mathbf{V}'_t$.

The target object type is represented as an integer in $\{0, 1, \ldots, T\}$, where T is the number of target object types used during training. We use an embedding of t to obtain a 32-dimensional vector. The vector is resized to be a $32 \times 1 \times 1$ tensor. The tensor is then expanded to be of size $32 \times 7 \times 7$, to form our goal target object type embedding $\mathbf{G}_t$, where the $32 \times 1 \times 1$ tensor is copied 7×7 times.

We concatenate $\mathbf{V}'_t$ and $\mathbf{G}_t$ to form a $64 \times 7 \times 7$ tensor, which is compressed with a 2-layer CNN to form a $32 \times 7 \times 7$ tensor, $\mathbf{Z}_t$. The tensor $\mathbf{Z}$ is flattened to form a 1568 dimensional vector, $\mathbf{z}_t$. Following [75], we use an embedding of the previous action, represented as an integer in $\{0, 1, \ldots, 5\}$, to obtain a 6 dimensional vector $\mathbf{a}_{t-1}$. We concatenate $\mathbf{z}_t$ and $\mathbf{a}_{t-1}$ to form a 1574 dimensional vector $\mathbf{x}_t$. The vector $\mathbf{x}_t$ is passed through a 1-layer GRU [17, 18] with a hidden belief state $\mathbf{b}_{t-1}$, of size 512, to obtain $\mathbf{b}_t$.

Using an actor-critic formulation, the 512-dimensional belief state $\mathbf{b}_t$ is passed through a 1-linear layer, representing the *actor*, to get a 6-dimensional vector, where each entry represents an action. The 6-dimensional vector is passed through a softmax function to obtain the agent's policy π (*i.e.* the probability distribution over the action space). We sample from π to choose the next action. We also pass the belief state $\mathbf{b}_t$ through a separate 1-linear layer, representing the *critic* to obtain the scalar v, estimating the value of the current state.

The "ProcTHOR+Large" is similar to the above except we: (1) use the larger RN50x16 CLIP-ResNet model, (2) use a 1024-dimensional hidden belief state in our GRU, and (3) input images to the model at a 512×384 resolution.

Hyperparameter	**Value**
Discount factor (γ)	0.99
GAE parameter (λ)	0.95
Value loss coefficient	0.5
Entropy loss coefficient	0.01
Clip parameter (ϵ)	0.1
Rollout timesteps	20
Rollouts per minibatch	1
Learning rate	3e-4
Optimizer	Adam [53]
Gradient clip norm	0.5

Table 4: Training hyperparameters for ObjectNav experiments.

Training. Each agent is trained using DD-PPO [90, 100], using a clip parameter $\epsilon = 0.1$, an entropy loss coefficient of 0.01, and a value loss coefficient of 0.5. Agents are trained to maximize the cumulative discounted rewards $\sum_{t=0}^{H} \gamma^t \cdot r_t$, where we set the discount factor γ to 0.99 and the episode's horizon H to 500 steps. We also employ GAE [89] parameterized by $\lambda = 0.95$.

Reward. The reward function follows that of [51]. Specifically, at each time step, it is calculated as $r_t = \max(0, \min \Delta_{0:t-1} - \Delta_t) + s_t - \rho$, where:

- $\min \Delta_{0:t-1}$ is the minimum L2 distance from the agent to any of the reachable instances of the target object type that the agent has observed over steps $\{0, 1, \ldots, t-1\}$.
- Δ_t is the current L2 distance from the agent to the nearest reachable instance of the target object type.
- s_t is the reward for successfully completing the episode. If the agent takes the DONE action and the episode is deemed successful, then s_t is 10. Otherwise, it is 0.

- ρ is the step penalty that encourages the agent to finish the episode quickly. It is set to 0.01.

ProcTHOR pre-training. We pre-train our ObjectNav agents on the full set of 10k training houses in PROCTHOR-10K.[1] We pre-train with all $T = 16$ target object types, which are shown in Table 5. The agent is trained for 423 million steps, although by 200 million steps, the agent has reached 90% of its peak performance. We used multi-node training to train on 3 AWS g4dn.12xlarge machines, which takes approximately 5 days to complete.

Object Type	RoboTHOR	HM3D-Semantics	AI2-iTHOR	ARCHITECTHOR
Alarm Clock	✓	✗	✓	✓
Apple	✓	✗	✓	✓
Baseball Bat	✓	✗	✓	✓
Basketball	✓	✗	✓	✓
Bed	✗	✓	✓	✓
Bowl	✓	✗	✓	✓
Chair	✗	✓	✓	✓
Garbage Can	✓	✗	✓	✓
House Plant	✓	✓	✓	✓
Laptop	✓	✗	✓	✓
Mug	✓	✗	✓	✓
Sofa	✗	✓	✓	✓
Spray Bottle	✓	✗	✓	✓
Television	✓	✓	✓	✓
Toilet	✗	✓	✓	✓
Vase	✓	✗	✓	✓

Table 5: The target objects that are used for each ObjectNav task.

Sampling target object types. To sample the target object type for a given episode, we restrict ourselves to only sampling target object types that have a possibility of leading to a successful episode. For instance, even if there is an object like an apple in the scene, it might be located in the fridge, and so if it was used as a target object, the agent would never succeed because the object would never appear visible in the frame (without any manipulation actions). Therefore, we impose a constraint that the target object must be visible without any form of manipulation.

For each house, we use an approximation to determine the set of target object instances that the agent can successfully reach, without any manipulation. Specifically, we start by teleporting the agent into the house, and then perform a BFS over a 0.25×0.25 meter grid to obtain the reachable positions in the scene. A position is considered reachable if teleporting to it would not cause any collisions with any other objects, and the agent is successfully placed on the floor. Then, for each candidate instance of every target object type, we look at the nearest 6 reachable agent positions $\langle x^{(a)}, z^{(a)} \rangle$ to the candidate object instance's center position. For each reachable agent position, we perform a raycast from the agent's camera height $y^{(a)}$ to up to 6 random *visibility points* on the object $\langle x^{(o)}, y^{(o)}, z^{(o)} \rangle$. Each object is annotated with visibility points, which are used as a fast approximation to determine if an object is visible with just using a few raycasts, instead of using full segmentation masks. If any of the raycasts from the agent's reachable position to the object's visibility point do not have any collisions with other objects (*e.g.* the raycast does not collide with the outside of the fridge), and the L2 distance between $\langle x^{(o)}, y^{(o)}, z^{(o)} \rangle$ and $\langle x^{(a)}, y^{(a)}, z^{(a)} \rangle$ is less than 1 meter, then the object instance is considered successfully reachable by the agent.

To choose a target object type, we use an ϵ-greedy sampling method. Specifically, with a probability of $\epsilon = 0.2$, we randomly sample a target object type that has at least 1 reachable object instance in a

[1] When training the "ProcTHOR+Large" model used in the HM3D challenge, we use a modified set of 10K houses, see below for details.

given house. With a probability of $1-\epsilon$, the target object type is the target object type that has been most infrequently sampled in the training process. Since some objects appear much more frequently than others (*e.g.* beds appear in many more houses than baseball bats), sampling based on the least commonly sampled target object types allows us to maintain a more uniform distribution of sampled target object types.

RoboTHOR. RoboTHOR is evaluated in both a 0-shot and fine-tuned setting. For 0-shot, we take the pre-trained model on PROCTHOR-10K and run it on the RoboTHOR evaluation tasks. For fine-tuning, we reduce T to the 12 RoboTHOR target object types, shown in Table 5 and train on the 60 provided training scenes. We fine-tune for 29 million steps, before validation performance starts to go down, on a machine with 8 NVIDIA Quadro RTX 8000 GPUs. Fine-tuning took about 7 hours to complete.

HM3D-Semantics. We evaluate on HM3D-Semantics in both a 0-shot and fine-tuned setting using the "ProcTHOR" and "ProcTHOR+Large" architectures described above, these two architectures have slightly different pretraining strategies.

"ProcTHOR" model. For 0-shot, we take the pre-trained model on PROCTHOR-10K, and run it on the HM3D-Semantics evaluation tasks. For fine-tuning, we reduce T to the 6 target object types used in HM3D-Semantics (see Table 5) and train on the 80 provided training houses. We use an early checkpoint from PROCTHOR pre-training, specifically from after 220 million steps. We performed fine-tuning on a machine with 8 NVIDIA RTX A6000 GPUs for approximately 220M steps, which took about 43 hours to complete.

"ProcTHOR+Large" model. We pre-train this model using PROCTHORLARGE-10K a variant of PROCTHOR-10K with houses sampled to better align to the distribution of houses in HM3D. In particular, PROCTHORLARGE-10K contains 10K procedurally generated houses each of which contains between 4 and 10 rooms (houses in PROCTHORLARGE-10K thus tend to be much larger than houses in PROCTHOR-10K). Moreover, during pretraining we only train our agent to navigate to the 6 object categories used in HM3D-Semantics. Fine-tuning is done identically as above. We use an early checkpoint from PROCTHOR pre-training, specifically from after 125 million steps. We performed fine-tuning on a machine with 8 NVIDIA RTX A6000 GPUs for approximately 185M steps taking 85 hours to complete.

AI2-iTHOR. Similar to RoboTHOR and HM3D-Semantics, we use AI2-iTHOR for both 0-shot and fine-tuning. For 0-shot, we take the pre-trained model on PROCTHOR-10K, and run it on the AI2-iTHOR evaluation tasks. Since the AI2-iTHOR evaluation tasks use the full set of target objects used during PROCTHOR pre-training, we do not need to update T. For fine-tuning, we use a machine with 8 TITAN V GPUs. We fine-tune for approximately 2 million steps before validation performance starts to go down, which takes about 1.5 hours to complete.

ArchitecTHOR. Since ARCHITECTHOR does not include any training scenes, we only use it for evaluation of the PROCTHOR pre-trained model. As shown in Table 5, ARCHITECTHOR evaluation uses the full-set of target object types that are used during PROCTHOR pre-training.

F.2 ArmPointNav experiments

In ArmPointNav, we followed the same architecture as [25]. The task is to move a target object from a starting location to a goal location using the relative location of the target in the agent's coordinate frame. The visual input is encoded using 3 convolutional layers followed by a linear layer to obtain a 512 feature vector. The 3D relative coordinates, specifying the targets, are embedded using three linear layers to a 512 embedding which combined with the visual encoding is input to the GRU. The agent is allowed to take up to 200 steps or the episode will automatically fail.

ProcTHOR pre-training. We pre-train our model on a subset of 7000 houses, on 58 object categories. For each episode, we move the agent to a random location, randomly choose an object in the room that is pickupable, and randomly select a target location. We train our model for 100M frames, running on 4 AWS g4dn.12xlarge machines. Running on a total of 16 GPUs and 192 CPU cores took 3 days of training. Table 6 shows the hyperparameters used for pre-training.

Hyperparameter	Value
Learning rate	3e-4
Gradient steps	128
Discount factor (γ)	0.99
GAE parameter (λ)	0.95
Gradient clip norm	0.5
Rotation Degrees	45
Step penalty	-0.01
Number of RNN Layers	1
Rollouts per minibatch	1
Optimizer	Adam [53]

Table 6: Training hyperparameters for ArmPointNav experiments.

AI2-iTHOR evaluation. We evaluate our model on 20 test rooms of AI2-THOR (5 kitchens, 5 living rooms, 5 bedrooms, 5 bathrooms), on a subset of 28 object categories for a total of 528 tasks. We attempted to perform fine-tuning on AI2-iTHOR, but none of the fine-tuning models performed better than the zero-shot model trained with PROCTHOR pre-training.

F.3 Rearrangement experiments

Following [98, 51], we use imitation learning (IL) to train all models for the 1-phase modality of the task. We divide the full training of the final model into two stages: pre-training in PROCTHOR and fine-tuning in AI2-iTHOR.

Hyperparameter	Value
Rollout timesteps	64
Batch size	7,680
Learning rate	$7.4 \cdot 10^{-4}$
Optimizer	Adam [53]
Gradient clip norm	0.5
$\text{BC}^{\text{tf}=1}$ steps	200,000
DAgger steps	2,000,000

Table 7: ProcTHOR pre-training hyperparameters for Rearrange experiments.

ProcTHOR pre-training. We pre-train our model on a subset of 2,500 one and two-room PROCTHOR-10K houses where a number of 1 to 5 objects are shuffled from their target poses in each episode, including two shuffle modalities: different openness degree (at most one object in an episode) and a different location (up to five objects in an episode). For each house, 20 episodes are sampled such that all shuffled objects are in the same room where the agent is initially spawned. We train with $2 \cdot 10^5$ steps of teacher forcing and 2 million steps of dataset aggregation [87], followed by about 180 million steps of behavior cloning. We use a small set of 200 episodes sampled from 20 validation houses unseen during training to select a checkpoint to evaluate every 5 million steps.

Running on 6 AWS g4dn.12xlarge (totaling 24 GPUs and 288 virtual CPU cores), pre-training with 240 parallel simulations took 4 days. Table 7 shows the hyperparameters used during pre-training.

AI2-iTHOR fine-tuning. We use the training dataset provided by [2] (4,000 episodes over 80 single-room scenes), and a small subset of 200 episodes from the also provided full validation set to perform model selection. We fine-tune for 3 million steps with 64-step long rollouts, 6 additional million steps with 96-step long rollouts, and another 6 million steps with 128-step long rollouts.

Running on 8 Titan X GPUs and 56 virtual CPU cores, fine-tuning with 40 parallel simulations took 16 hours.

	Navigation FPS		Isolated Interaction FPS		Environment Query FPS	
Compute	AI2-iTHOR	RoboTHOR	AI2-iTHOR	RoboTHOR	AI2-iTHOR	RoboTHOR
8 GPUs	5,779±189	9,195±294	5,411±190	6,331±137	463,446±18,577	412,550±21,806
1 GPU	1,316±19	1,648±11	1,451±72	1,539±5	169,092±4,232	163,660±3,336
1 Process	180±9	340±26	141±2	217±1	15,584±156	15,578±164
	PROCTHOR-S	PROCTHOR-L	PROCTHOR-S	PROCTHOR-L	PROCTHOR-S	PROCTHOR-L
8 GPUs	8,599±359	3,208±127	6,488±250	2,861±107	480,205±19,684	433,587±18,729
1 GPU	1,427±74	6,280±40	1,265±71	597±37	160,622±2,846	157,567±2,689
1 Process	240±69	115±19	180±42	93±15	14,825±199	14,916±186

Table 8: Comparing performance benchmarks in PROCTHOR to baselines in AI2-iTHOR and RoboTHOR. FPS for navigation, interaction, and querying the environment for data. PROCTHOR-S and PROCTHOR-L denotes small and large PROCTHOR houses, respectively.

G Performance Benchmark

To calculate the FPS performance benchmark shown in the Analysis section, we partitioned houses into small houses (1-3 room houses) and large houses (7-10 room houses). For the navigation benchmark, we perform move and rotate actions. For the interaction benchmark, we performing a pushing object action. For querying the environment for data, we obtain a piece of metadata from the environment that is not commonly provided at each time step (*e.g.* checking the dimensions of the agent). At each time step, we render a single $3 \times 224 \times 224$ RGB image from the agent's egocentric perspective. Experiments were conducted on a server with 8 NVIDIA Quadro RTX 8000 GPUs. We employ 15 processes for the single GPU tests and 120 processes for the 8 GPU tests, evenly divided across the GPUs. Table 8 shows the comparisons to AI2-iTHOR and RoboTHOR.

H Robustness

We ran ProcTHOR ObjectNav pre-training with 5 different random seeds for 100M steps and found that the variance across seeds is quite small. This measurement was performed for our 0-shot results on a set of 1000 ObjectNav tasks divided evenly between unseen ProcTHOR validation homes, ArchitecTHOR validation scenes, AI2-iTHOR validation scenes, and RoboTHOR validation scenes. Here, we obtained similar performance across all run which had a train success of $67.87\% \pm 2.89\%$ and a val success of $45.3\% \pm 1.2\%$.

I Broader Impact

This work focuses on increasing the generalization abilities of robotic agents on various tasks. We specifically focus on robots that operate in household environments. More capable robotic agents can help improve the lives of many by assisting with cooking, cleaning, and providing social interaction. Furthermore, robots can provide a wide range of health benefits. For example, they could give domestic assistance to individuals with physical and mental disabilities and the elderly. They could provide social and emotional support to children, adolescents, and adults, such as delivering personalized educational content, reducing loneliness, and counseling in times of crisis. We can also use home-assisted robots to monitor and provide feedback on people's physical activity, sleep, and diet.

However, the adoption of home-assisted robots could have several undesirable social consequences. One is that home-assisted social robots may lead individuals to become more dependant on robots for companionship and care, leading to increased social isolation and loneliness. Another concern is that they may exacerbate existing inequities, as those who can afford to buy and maintain robots will have access to care and assistance that those who cannot will not. Furthermore, because robots would have access to sensitive information about people's daily lives, they could threaten privacy and security. Finally, robots have the potential to be exploited for malicious intent, such as for mass surveillance or being used for autonomous warfare. As a community, we need to work to reduce the risks of social robots while maximizing the benefits for the common good.

J Contributions

Matt Deitke designed and implemented the procedure to generate houses, implemented ObjectNav pre-training experiments and fine-tuning experiments, built the website, advised and implemented parts of the Unity backend, built the platform to visualize assets and create semantic asset groups, contributed to visuals, and wrote the paper.

Eli VanderBilt standardized AI2-THOR's asset and material database to make it usable with PROCTHOR, led the development of ARCHITECTHOR, implemented parts of the Unity backend, created new 3D assets and skyboxes, advised on lighting the houses, and contributed to visuals.

Alvaro Herrasti led the Unity backend development that creates a house from a JSON specification.

Luca Weihs advised the work on experiments, assisted with rearragement experiments, implemented ObjectNav fine-tuning on HM3D-Semantics, and wrote parts of the paper.

Jordi Salvador implemented rearrangement experiments, advised on multi-node training experiments, and wrote parts of the paper.

Kiana Ehsani implemented ArmPointNav experiments and wrote parts of the paper.

Winson Han implemented parts of ARCHITECTHOR, implemented parts of the Unity backend, and contributed to visuals.

Eric Kolve advised on the Unity backend development.

Ali Farhadi advised on the research direction.

Aniruddha Kembhavi advised on research direction, the ARCHITECTHOR development, and the house generation process and wrote the paper.

Roozbeh Mottaghi advised on the research direction, the Unity backend, the ARCHITECTHOR development, and the house generation process and wrote the paper.

References

[1] Adam, Anuj Mahajan, Catarina Barros, Charlie Deck, Jakob Bauer, Jakub Sygnowski, Maja Trebacz, Max Jaderberg, Michael Mathieu, Nat McAleese, Nathalie Bradley-Schmieg, Nathaniel Wong, Nicolas Porcel, Roberta Raileanu, Steph Hughes-Fitt, Valentin Dalibard, and Wojciech Marian Czarnecki. Open-ended learning leads to generally capable agents. *arXiv*, 2021. 24, 27

[2] Allen Institute for AI. Rearrangement Challenge 2022. https://leaderboard.allenai.org/ithor_rearrangement_1phase_2022. 39

[3] Peter Anderson, Angel X. Chang, Devendra Singh Chaplot, Alexey Dosovitskiy, Saurabh Gupta, Vladlen Koltun, Jana Kosecka, Jitendra Malik, Roozbeh Mottaghi, Manolis Savva, and Amir Roshan Zamir. On evaluation of embodied navigation agents. *arXiv*, 2018. 35

[4] Scott A Arvin and Donald H House. Modeling architectural design objectives in physically based space planning. *Automation in Construction*, 11(2):213–225, 2002. 23

[5] Dhruv Batra, Aaron Gokaslan, Aniruddha Kembhavi, Oleksandr Maksymets, Roozbeh Mottaghi, Manolis Savva, Alexander Toshev, and Erik Wijmans. Objectnav revisited: On evaluation of embodied agents navigating to objects. *arXiv*, 2020. 35

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *NeurIPS*, 2020. 24

[7] Carnegie Mellon University. Locobot: an open source low cost robot. http://www.locobot.org/. 35

[8] Angel X Chang, Mihail Eric, Manolis Savva, and Christopher D Manning. Sceneseer: 3d scene design with natural language. *arXiv preprint arXiv:1703.00050*, 2017. 24

[9] Angel X. Chang, Thomas A. Funkhouser, Leonidas J. Guibas, Pat Hanrahan, Qi-Xing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. Shapenet: An information-rich 3d model repository. *arXiv*, 2015. 24

[10] Angel X. Chang, Will Monroe, Manolis Savva, Christopher Potts, and Christopher D. Manning. Text to 3d scene generation with rich lexical grounding. In *ACL*, 2015. 23, 24

[11] Angel X. Chang, Manolis Savva, and Christopher D. Manning. Learning spatial knowledge for text to 3d scene generation. In *EMNLP*, 2014. 23, 24

[12] Jorge L. Charco, Angel Domingo Sappa, Boris Xavier Vintimilla, and Henry O. Velesaca. Camera pose estimation in multi-view environments: From virtual scenarios to the real world. *Image Vis. Comput.*, 2021. 27

[13] Aditya Chattopadhyay, Xi Zhang, David Paul Wipf, Rene Vidal, and Himanshu Arora. Structured graph variational autoencoders for indoor furniture layout generation. *arXiv preprint arXiv:2204.04867*, 2022. 24

[14] Changan Chen, Ziad Al-Halah, and Kristen Grauman. Semantic audio-visual navigation. In *CVPR*, 2021. 27

[15] Changan Chen, Unnat Jain, Carl Schissler, Sebastia Vicenc Amengual Gari, Ziad Al-Halah, Vamsi Krishna Ithapu, Philip Robinson, and Kristen Grauman. Soundspaces: Audio-visual navigation in 3d environments. In *ECCV*, 2020. 27

[16] Rohan Chitnis, Tom Silver, Joshua B. Tenenbaum, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Learning Neuro-Symbolic Relational Transition Models for Bilevel Planning. *arXiv*, 2021. 27

[17] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *EMNLP*, 2014. 36

[18] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv*, 2014. 36

[19] Jasmine Collins, Shubham Goel, Kenan Deng, Achleshwar Luthra, Leon Xu, Erhan Gundogdu, Xi Zhang, Tomas F Yago Vicente, Thomas Dideriksen, Himanshu Arora, Matthieu Guillaumin, and Jitendra Malik. Abo: Dataset and benchmarks for real-world 3d object understanding. In *CVPR*, 2022. 24

[20] Matt Deitke, Winson Han, Alvaro Herrasti, Aniruddha Kembhavi, Eric Kolve, Roozbeh Mottaghi, Jordi Salvador, Dustin Schwenk, Eli VanderBilt, Matthew Wallingford, Luca Weihs, Mark Yatskar, and Ali Farhadi. Robothor: An open simulation-to-real embodied ai platform. In *CVPR*, 2020. 27, 35

[21] Matt Deitke, Aniruddha Kembhavi, and Luca Weihs. PRIOR: A Python Package for Seamless Data Distribution in AI Workflows, 2022. 27

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019. 24

[23] Laura Downs, Anthony Francis, Nate Koenig, Brandon Kinman, Ryan Hickman, Krista Reymann, Thomas B McHugh, and Vincent Vanhoucke. Google scanned objects: A high-quality dataset of 3d scanned household items. In *ICRA*, 2022. 24

[24] Kiana Ehsani, Ali Farhadi, Aniruddha Kembhavi, and Roozbeh Mottaghi. Object manipulation via visual target localization. *arXiv*, 2022. 27

[25] Kiana Ehsani, Winson Han, Alvaro Herrasti, Eli VanderBilt, Luca Weihs, Eric Kolve, Aniruddha Kembhavi, and Roozbeh Mottaghi. ManipulaTHOR: A Framework for Visual Object Manipulation. In *CVPR*, 2021. 27, 38

[26] Kiana Ehsani, Roozbeh Mottaghi, and Ali Farhadi. Segan: Segmenting and generating the invisible. In *CVPR*, 2018. 27

[27] Di Feng, Christian Haase-Schuetz, Lars Rosenbaum, Heinz Hertlein, Fabian Duffhauss, Claudius Gläser, Werner Wiesbeck, and Klaus C. J. Dietmayer. Deep multi-modal object detection and semantic segmentation for autonomous driving: Datasets, methods, and challenges. *IEEE Trans. on Intelligent Transportation Systems*, 2021. 27

[28] Matthew Fisher, Daniel Ritchie, Manolis Savva, Thomas Funkhouser, and Pat Hanrahan. Example-based synthesis of 3d object arrangements. *ACM Transactions on Graphics (TOG)*, 31(6):1–11, 2012. 23, 24

[29] Matthew Fontaine and Stefanos Nikolaidis. A quality diversity approach to automatically generating human-robot interaction scenarios in shared autonomy. *arXiv preprint arXiv:2012.04283*, 2020. 24

[30] Jonas Freiknecht and Wolfgang Effelsberg. Procedural generation of multistory buildings with interior. *IEEE Transactions on Games*, 12(3):323–336, 2019. 23

[31] Huan Fu, Bowen Cai, Lin Gao, Ling-Xiao Zhang, Jiaming Wang, Cao Li, Qixun Zeng, Chengyue Sun, Rongfei Jia, Binqiang Zhao, et al. 3d-front: 3d furnished rooms with layouts and semantics. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10933–10942, 2021. 23, 24

[32] Huan Fu, Rongfei Jia, Lin Gao, Mingming Gong, Binqiang Zhao, Steve Maybank, and Dacheng Tao. 3d-future: 3d furniture shape with texture. *International Journal of Computer Vision*, 129(12):3313–3337, 2021. 24

[33] Samir Yitzhak Gadre, Kiana Ehsani, Shuran Song, and Roozbeh Mottaghi. Continuous scene representations for embodied ai. In *CVPR*, 2022. 27

[34] Chuang Gan, Yiwei Zhang, Jiajun Wu, Boqing Gong, and Joshua B Tenenbaum. Look, listen, and act: Towards audio-visual embodied navigation. In *ICRA*, 2020. 27

[35] Chuang Gan, Siyuan Zhou, Jeremy Schwartz, Seth Alter, Abhishek Bhandwaldar, Dan Gutfreund, Daniel LK Yamins, James J DiCarlo, Josh McDermott, Antonio Torralba, et al. The threedworld transport challenge: A visually guided task-and-motion planning benchmark for physically realistic embodied ai. *arXiv*, 2021. 27

[36] Timnit Gebru, Jamie H. Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna M. Wallach, Hal Daumé, and Kate Crawford. Datasheets for datasets. *Comm. of the ACM*, 2021. 28, 32

[37] Tobias Germer and Martin Schwarz. Procedural arrangement of furniture for real-time walkthroughs. In *Computer Graphics Forum*, volume 28, pages 2068–2078. Wiley Online Library, 2009. 23, 24

[38] Daniel Gordon, Aniruddha Kembhavi, Mohammad Rastegari, Joseph Redmon, Dieter Fox, and Ali Farhadi. Iqa: Visual question answering in interactive environments. In *CVPR*, 2018. 27

[39] Klaus Greff, Francois Belletti, Lucas Beyer, Carl Doersch, Yilun Du, Daniel Duckworth, David J Fleet, Dan Gnanapragasam, Florian Golemo, Charles Herrmann, Thomas Kipf, Abhijit Kundu, Dmitry Lagun, Issam Laradji, Hsueh-Ti (Derek) Liu, Henning Meyer, Yishu Miao, Derek Nowrouzezahrai, Cengiz Oztireli, Etienne Pot, Noha Radwan, Daniel Rebain, Sara Sabour, Mehdi S. M. Sajjadi, Matan Sela, Vincent Sitzmann, Austin Stone, Deqing Sun, Suhani Vora, Ziyu Wang, Tianhao Wu, Kwang Moo Yi, Fangcheng Zhong, and Andrea Tagliasacchi. Kubric: a scalable dataset generator. In *CVPR*, 2022. 27

[40] Padraig Higgins, Ryan Barron, and Cynthia Matuszek. Head pose as a proxy for gaze in virtual reality. In *Workshop on Virtual, Augmented, and Mixed Reality for HRI*, 2022. 27

[41] Md Shahadat Hossain and Abdus Salam. Text-to-3d scene generation using semantic parsing and spatial knowledge with rule based system. *International Journal of Computer Science Issues (IJCSI)*, 14(5):37–41, 2017. 23, 24

[42] Ruizhen Hu, Zeyu Huang, Yuhan Tang, Oliver Matias van Kaick, Hao Zhang, and Hui Huang. Graph2plan: Learning floorplan generation from layout graphs. *ACM Trans. on Graphics*, 2020. 23

[43] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. *arXiv*, 2022. 27

[44] Unnat Jain, Luca Weihs, Eric Kolve, Ali Farhadi, Svetlana Lazebnik, Aniruddha Kembhavi, and Alexander Schwing. A cordial sync: Going beyond marginal policies for multi-agent embodied tasks. In *ECCV*, 2020. 27

[45] Unnat Jain, Luca Weihs, Eric Kolve, Mohammad Rastegari, Svetlana Lazebnik, Ali Farhadi, Alexander G Schwing, and Aniruddha Kembhavi. Two body problem: Collaborative visual task completion. In *CVPR*, 2019. 27

[46] Abhishek Kadian, Joanne Truong, Aaron Gokaslan, Alexander Clegg, Erik Wijmans, Stefan Lee, Manolis Savva, Sonia Chernova, and Dhruv Batra. Sim2real predictivity: Does evaluation in simulation predict real-world performance? *IEEE Robotics and Automation Letters*, 2020. 27, 35

[47] Peter Kán and Hannes Kaufmann. Automatic furniture arrangement using greedy cost minimization. In *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pages 491–498. IEEE, 2018. 23, 24

[48] Yash Kant, Arun Ramachandran, Sriram Yenamandra, Igor Gilitschenski, Dhruv Batra, Andrew Szot, and Harsh Agrawal. Housekeep: Tidying virtual households using commonsense reasoning. *arXiv preprint arXiv:2205.10712*, 2022. 24

[49] Amlan Kar, Aayush Prakash, Ming-Yu Liu, Eric Cameracci, Justin Yuan, Matt Rusiniak, David Acuna, Antonio Torralba, and Sanja Fidler. Meta-sim: Learning to generate synthetic datasets. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 4550–4559, 2019. 24

[50] Siddharth Karamcheti, Dorsa Sadigh, and Percy Liang. Learning adaptive language interfaces through decomposition. *arXiv*, 2020. 27

[51] Apoorv Khandelwal, Luca Weihs, Roozbeh Mottaghi, and Aniruddha Kembhavi. Simple but effective: Clip embeddings for embodied ai. In *CVPR*, 2021. 27, 35, 36, 39

[52] Seung Wook Kim, Yuhao Zhou, Jonah Philion, Antonio Torralba, and Sanja Fidler. Learning to simulate dynamic environments with gamegan. In *CVPR*, 2020. 27

[53] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv*, 2014. 36, 39

[54] Jing Yu Koh, Harsh Agrawal, Dhruv Batra, Richard Tucker, Austin Waters, Honglak Lee, Yinfei Yang, Jason Baldridge, and Peter Anderson. Simple and effective synthesis of indoor 3d scenes. *arXiv*, 2022. 27

[55] Jing Yu Koh, Honglak Lee, Yinfei Yang, Jason Baldridge, and Peter Anderson. Pathdreamer: A world model for indoor navigation. In *ICCV*, 2021. 27

[56] Klemen Kotar and Roozbeh Mottaghi. Interactron: Embodied adaptive object detection. In *CVPR*, 2022. 27

[57] Jacob Krantz, Erik Wijmans, Arjun Majumdar, Dhruv Batra, and Stefan Lee. Beyond the nav-graph: Vision-and-language navigation in continuous environments. In *ECCV*, 2020. 27

[58] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. Rma: Rapid motor adaptation for legged robots. In *RSS*, 2021. 27

[59] Manyi Li, Akshay Gadi Patil, Kai Xu, Siddhartha Chaudhuri, Owais Khan, Ariel Shamir, Changhe Tu, Baoquan Chen, Daniel Cohen-Or, and Hao Zhang. Grains: Generative recursive autoencoders for indoor scenes. *ACM Trans. on Graphics*, 2019. 24

[60] Yunzhu Li, Shuang Li, Vincent Sitzmann, Pulkit Agrawal, and Antonio Torralba. 3d neural scene representations for visuomotor control. In *CoRL*, 2022. 27

[61] Ricardo Lopes, Tim Tutenel, Ruben M Smelik, Klaas Jan De Kraker, and Rafael Bidarra. A constrained growth method for procedural floor plan generation. In *Game-ON*, 2010. 10, 23

[62] Andrew Luo, Zhoutong Zhang, Jiajun Wu, and Joshua B Tenenbaum. End-to-end optimization of scene layout. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3754–3763, 2020. 24

[63] Haokuan Luo, Albert Yue, Zhang-Wei Hong, and Pulkit Agrawal. Stubborn: A strong baseline for indoor object navigation. *arXiv*, 2022. 27

[64] Rui Ma, Akshay Gadi Patil, Matthew Fisher, Manyi Li, Sören Pirk, Binh-Son Hua, Sai-Kit Yeung, Xin Tong, Leonidas Guibas, and Hao Zhang. Language-driven synthesis of 3d scenes from scene databases. *ACM Transactions on Graphics (TOG)*, 37(6):1–16, 2018. 23, 24

[65] Fernando Marson and Soraia Raupp Musse. Automatic real-time generation of floor plans based on squarified treemaps algorithm. *International Journal of Computer Games Technology*, 2010. 9, 23

[66] Cristina Mata, Nick Locascio, Mohammed Azeem Sheikh, Kenny Kihara, and Dan Fischetti. Standardsim: A synthetic dataset for retail environments. In *ICIAP*, 2022. 27

[67] Paul Merrell, Eric Schkufza, Zeyang Li, Maneesh Agrawala, and Vladlen Koltun. Interactive furniture layout using interior design guidelines. *ACM transactions on graphics (TOG)*, 30(4):1–10, 2011. 23

[68] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020. 27

[69] Kaichun Mo, Shilin Zhu, Angel X. Chang, Li Yi, Subarna Tripathi, Leonidas J. Guibas, and Hao Su. PartNet: A large-scale benchmark for fine-grained and hierarchical part-level 3D object understanding. In *CVPR*, 2019. 24

[70] Mark Murnane, Padraig Higgins, Monali Saraf, Francis Ferraro, Cynthia Matuszek, and Don Engel. A simulator for human-robot interaction in virtual reality. In *VRW*, 2021. 27

[71] Yashraj Narang, Kier Storey, Iretiayo Akinola, Miles Macklin, Philipp Reist, Lukasz Wawrzyniak, Yunrong Guo, Adam Moravanszky, Gavriel State, Michelle Lu, Ankur Handa, and Dieter Fox. Factory: Fast contact for robotic assembly. In *RSS*, 2022. 27

[72] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *J. Mach. Learn. Res.*, 21:181:1–181:50, 2020. 24

[73] Nelson Nauata, Kai-Hung Chang, Chin-Yi Cheng, Greg Mori, and Yasutaka Furukawa. House-gan: Relational generative adversarial networks for graph-constrained house layout generation. In *European Conference on Computer Vision*, pages 162–177. Springer, 2020. 23

[74] Nelson Nauata, Sepidehsadat Hosseini, Kai-Hung Chang, Hang Chu, Chin-Yi Cheng, and Yasutaka Furukawa. House-gan++: Generative adversarial layout refinement network towards intelligent computational agent for professional architects. In *CVPR*, 2021. 23

[75] Tianwei Ni, Kiana Ehsani, Luca Weihs, and Jordi Salvador. Towards disturbance-free visual mobile manipulation. *arXiv*, 2021. 27, 36

[76] OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas A. Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei M. Zhang. Solving rubik's cube with a robot hand. *ArXiv*, abs/1910.07113, 2019. 24

[77] Aishwarya Padmakumar, Jesse Thomason, Ayush Shrivastava, Patrick Lange, Anjali Narayan-Chen, Spandana Gella, Robinson Piramuthu, Gokhan Tur, and Dilek Hakkani-Tur. Teach: Task-driven embodied agents that chat. In *AAAI*, 2022. 27

[78] Wamiq Reyaz Para, Paul Guerrero, Niloy Mitra, and Peter Wonka. Cofs: Controllable furniture layout synthesis. *arXiv preprint arXiv:2205.14657*, 2022. 24

[79] Despoina Paschalidou, Amlan Kar, Maria Shugrina, Karsten Kreis, Andreas Geiger, and Sanja Fidler. Atiss: Autoregressive transformers for indoor scene synthesis. *Advances in Neural Information Processing Systems*, 34:12013–12026, 2021. 24

[80] Claudia Pérez-D'Arpino, Can Liu, Patrick Goebel, Roberto Martín-Martín, and Silvio Savarese. Robot navigation in constrained pedestrian environments using reinforcement learning. In *ICRA*, 2021. 27

[81] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *ICML*, 2021. 36

[82] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67, 2020. 24

[83] Ram Ramrakhya, Eric Undersander, Dhruv Batra, and Abhishek Das. Habitat-web: Learning embodied object-search strategies from human demonstrations at scale. In *CVPR*, 2022. 35

[84] Jeremy Reizenstein, Roman Shapovalov, Philipp Henzler, Luca Sbordone, Patrick Labatut, and David Novotny. Common objects in 3d: Large-scale learning and evaluation of real-life 3d category reconstruction. In *ICCV*, 2021. 24

[85] Daniel Ritchie, Kai Wang, and Yu-An Lin. Fast and flexible indoor scene synthesis via deep convolutional generative models. In *CVPR*, 2019. 24

[86] Alex Rodriguez and Alessandro Laio. Clustering by fast search and find of density peaks. *science*, 344(6191):1492–1496, 2014. 24

[87] Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, 2011. 39

[88] Julian F Rosser, Gavin Smith, and Jeremy G Morley. Data-driven estimation of building interior plans. *International Journal of Geographical Information Science*, 31(8):1652–1674, 2017. 23

[89] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *ICLR*, 2016. 36

[90] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv*, 2017. 36

[91] Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *CVPR*, 2020. 27

[92] Shuran Song, Fisher Yu, Andy Zeng, Angel X Chang, Manolis Savva, and Thomas Funkhouser. Semantic scene completion from a single depth image. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1746–1754, 2017. 24

[93] Sanjana Srivastava, Chengshu Li, Michael Lingelbach, Roberto Mart'in-Mart'in, Fei Xia, Kent Vainio, Zheng Lian, Cem Gokmen, S. Buch, C. Karen Liu, Silvio Savarese, Hyowon Gweon, Jiajun Wu, and Li Fei-Fei. Behavior: Benchmark for everyday household activities in virtual, interactive, and ecological environments. In *CoRL*, 2021. 27

[94] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, 2015. 36

[95] Matthew Tancik, Vincent Casser, Xinchen Yan, Sabeek Pradhan, Ben Mildenhall, Pratul P Srinivasan, Jonathan T Barron, and Henrik Kretzschmar. Block-nerf: Scalable large scene neural view synthesis. In *CVPR*, 2022. 27

[96] Kai Wang, Yu-An Lin, Ben Weissmann, Manolis Savva, Angel X Chang, and Daniel Ritchie. Planit: Planning and instantiating indoor scenes with relation graph and spatial prior networks. *ACM Transactions on Graphics (TOG)*, 38(4):1–15, 2019. 24

[97] Xinpeng Wang, Chandan Yeshwanth, and Matthias Nießner. Sceneformer: Indoor scene generation with transformers. In *3DV*, 2021. 24

[98] Luca Weihs, Matt Deitke, Aniruddha Kembhavi, and Roozbeh Mottaghi. Visual room rearrangement. In *CVPR*, 2021. 24, 27, 39

[99] Tomer Weiss, Alan Litteneker, Noah Duncan, Masaki Nakada, Chenfanfu Jiang, Lap-Fai Yu, and Demetri Terzopoulos. Fast and scalable position-based layout synthesis. *IEEE Transactions on Visualization and Computer Graphics*, 25(12):3231–3243, 2018. 23, 24

[100] Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. Dd-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames. In *ICLR*, 2019. 27, 36

[101] Mitchell Wortsman, Kiana Ehsani, Mohammad Rastegari, Ali Farhadi, and Roozbeh Mottaghi. Learning to learn how to learn: Self-adaptive visual navigation using meta-learning. In *CVPR*, 2019. 27

[102] Qi Wu, Cheng-Ju Wu, Yixin Zhu, and Jungseock Joo. Communicative learning with natural gestures for embodied navigation agents with human-in-the-scene. In *IROS*, 2021. 27

[103] Wenming Wu, Xiao-Ming Fu, Rui Tang, Yuhan Wang, Yu-Hao Qi, and Ligang Liu. Data-driven interior plan generation for residential buildings. *ACM Trans. on Graphics*, 2019. 23

[104] Fei Xia, Chengshu Li, Roberto Mart'in-Mart'in, Or Litany, Alexander Toshev, and Silvio Savarese. Relmogen: Integrating motion generation in reinforcement learning for mobile manipulation. In *ICRA*, 2021. 27

[105] Kun Xu, Kang Chen, Hongbo Fu, Wei-Lun Sun, and Shi-Min Hu. Sketch2scene: Sketch-based co-retrieval and co-placement of 3d models. *ACM Transactions on Graphics (TOG)*, 32(4):1–15, 2013. 24

[106] Samir Yitzhak Gadre, Mitchell Wortsman, Gabriel Ilharco, Ludwig Schmidt, and Shuran Song. Clip on wheels: Zero-shot object navigation as object localization and exploration. *arXiv*, 2022. 27

[107] Lap Fai Yu, Sai Kit Yeung, Chi Keung Tang, Demetri Terzopoulos, Tony F Chan, and Stanley J Osher. Make it home: automatic optimization of furniture arrangement. *ACM Transactions on Graphics (TOG)-Proceedings of ACM SIGGRAPH 2011, v. 30,(4), July 2011, article no. 86*, 30(4), 2011. 23, 24

[108] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022. 24

[109] Shao-Kui Zhang, Wei-Yu Xie, and Song-Hai Zhang. Geometry-based layout generation with hyper-relations among objects. *Graphical Models*, 116:101104, 2021. 23

[110] Zaiwei Zhang, Zhenpei Yang, Chongyang Ma, Linjie Luo, Alexander G. Huth, Etienne Vouga, and Qixing Huang. Deep generative modeling for scene synthesis via hybrid representations. *ACM Trans. on Graphics*, 2020. 24

[111] Kaiyu Zheng, Rohan Chitnis, Yoonchang Sung, George Konidaris, and Stefanie Tellex. Towards Optimal Correlational Object Search. In *ICRA*, 2022. 27

[112] Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J Lim, Abhinav Gupta, Li Fei-Fei, and Ali Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *ICRA*, 2017. 27