Embodied Question Answering: Supplementary Document

Abhishek Das[1*], Samyak Datta[1], Georgia Gkioxari[2], Stefan Lee[1], Devi Parikh[2,1], Dhruv Batra[2,1]
[1]Georgia Institute of Technology, [2]Facebook AI Research
[1]{abhshkdz, samyak, steflee}@gatech.edu [2]{gkioxari, parikh, dbatra}@fb.com
embodiedqa.org

Abstract

This supplementary document is organized as follows:

- *Sec. 1 presents the question-answer generation engine in detail, including functional programs associated with questions, and checks and balances in place to avoid ambiguities, biases, and redundancies.*
- *Sec. 2 describes the CNN models that serve as the vision module for our EmbodiedQA model. We describe the model architecture, along with the training details, quantitative as well as qualitative results.*
- *Sec. 3 describes the answering module in our agent.*
- *Sec. 4 reports machine question answering performance conditioned on human navigation paths (collected via human studies on Amazon Mechanical Turk).*
- *Finally, `embodiedqa.org` shows example navigation and answer predictions by our agent.*

1. Question-Answer Generation Engine

Recall that each question in EQA is represented as a functional program that can be executed on the environment yielding an answer. In this section, we describe this process in detail. In the descriptions that follow, an 'entity' can refer to either a queryable room or a queryable object from the House3D [1] environment.

Functional Forms of Questions. The functional programs are composed of elementary operations described below:

1. `select(entity)`: Fetches a list of entities from the environment. This operation is similar to the 'select' query in relational databases.
2. `singleton(entity)`: Performs filtering to retain entities that occur exactly once. For example, calling `singleton(rooms)` on the set of rooms [*'living_room'*, *'bedroom'*, *'bedroom'*] for a given house will return [*'living_room'*].
3. `blacklist(template)`: This function operates on a list of object entities and filters out a pre-defined list of objects that are blacklisted for the given template. We do not ask questions of a given template type corresponding to any of the blacklisted objects. For example, if the blacklist contains the objects {*'column'*, *'range_hood'*, *'toy'*} and the objects list that the function receives is {*'piano'*, *'bed'*, *'column'*}, then the output of the `blacklist(·)` function is: {*'piano'*, *'bed'*}
4. `query(template)`: This is used to generate the questions strings for the given template on the basis of the entities that it receives. For example, if `query(location)` receives the following set of object entities as input: [*'piano'*, *'bed'*, *'television'*], it produces 3 question strings of the form: *what room is the <OBJ> located in?* where *<OBJ>* = {*'piano'*, *'bed'*, *'television'*}.
5. `relate()`: This elementary operation is used in the functional form for preposition questions. Given a list of object entities, it returns a subset of the pairs of objects that have a {*'on'*, *'above'*, *'under'*, *'below'*, *'next to'*} spatial relationship between them.
6. `distance()`: This elementary operation is used in the functional form for distance comparison questions. Given a list of object entities, it returns triplets of objects such that the first object is closer/farther to the anchor object than the second object.

Having described the elementary operations that make up our functional forms, the explanations of the functional forms for each question template is given below. We categorize the question types into 3 categories based on the objects and the rooms that they refer to.

*Work partially done during an internship at Facebook AI Research.

Template	Functional Form
location	select(objects) → singleton(objects) → blacklist(location) → query(location)
color	select(objects) → singleton(objects) → blacklist(color) → query(color)
color_room	select(rooms) → singleton(rooms) → select(objects) → singleton(objects) → blacklist(color) → query(color_room)
preposition	select(rooms) → singleton(rooms) → select(objects) → singleton(objects) → blacklist(preposition) → relate() → query(preposition)
exist	select(rooms) → singleton(rooms) → select(objects) → blacklist(exist) → query(exist)
logical	select(rooms) → singleton(rooms) → select(objects) → blacklist(exist) → query(logical)
count	select(rooms) → singleton(rooms) → select(objects) → blacklist(count) → query(count)
room_count	select(rooms) → query(room_count)
distance	select(rooms) → singleton(rooms) → select(objects) → singleton(objects) → blacklist(distance) → distance() → query(distance)

Table 1: Functional forms of all question types in the EQA dataset

1. **Unambiguous Object**: There are certain question types that inquire about an object that must be unique and unambiguous throughout the environment. Examples of such question types are `location` and `color`. For example, we should ask *'what room is the piano located in?'* if there is only a single instance of a *'piano'* in the environment. Hence, the functional forms for location and color have the following structure:

   `select(objects)` → `singleton(objects)` → `query(location/color).`

   `select(objects)` gets the list of all objects in the house and then `singleton(objects)` only retains objects that occur once, thereby ensuring unambiguity. The `query(template)` function prepares the question string by filling in the slots in the template string.

2. **Unambiguous Room + Unambiguous Object**: In continuation of the above, there is another set of question types that talk about objects in rooms where in addition to the objects being unique, the rooms should also be unambiguous. Examples of such question types include `color_room`, `preposition`, and `distance`. The additional unambiguity constraint on the room is because the question *'what is next to the bathtub in the bathroom?'* would become ambiguous if there are two or more bathrooms in the house. The functional forms for such types are given by the following structure:

   `select(rooms)` → `singleton(rooms)` → `select(objects)` → `singleton(objects)` → `query(template).`

   The first two operations in the sequence result in a list of unambiguous rooms whereas the next two result in a list of unambiguous objects in those rooms. Note that when `select(·)` appears as the first operation in the sequence (*i.e.*, select operates on an empty list), it is used to fetch the set of rooms or objects across the entire house. However, in this case, `select(objects)` operates on a set of rooms (the output of `select(rooms)` → `singleton(rooms)` →), so it returns the set of objects found in those specific rooms (as opposed to fetching objects across all rooms in the house).

3. **Unambiguous Room**: The final set of question types are the ones where the rooms need to be unambiguous, but the objects in those rooms that are being referred to do not. Examples of such question types are: `existence`, `logical`, and `count`. It is evident that for asking about the existence of objects or while counting them, we do not require the object to have only a single instance. *'Is there a television in the living room?'* is a perfectly valid question, even if there are multiple televisions in the living room (provided that there is a single living room in the house). The template structure for this is a simplified version of (2):

   `select(rooms)` → `singleton(rooms)` → `select(objects)` → `query(template).`

   Note that we have dropped `singleton(objects)` from the sequence as we no longer require that condition to hold true.

See Table 1 for a complete list of question functional forms.

Checks and balances. Since one of our goals is to benchmark performance of our agents with human performance, we want to avoid questions that are cumbersome for a human to navigate for or to answer. Additionally, we would also like to have a balanced distribution of answers so that the agent is not able to simply exploit dataset biases and answer questions effectively without exploration. This section describes in detail the various checks that we have in place to ensure these properties.

1. **Entropy+Frequency-based Filtering**: It is important to ensure that the distribution of answers for a question is not too 'peaky', otherwise the mode guess from this distribution will do unreasonably well as a baseline. Thus, we compute the normalized entropy of the distribution of answers for a question. We drop questions where the normalized entropy is below 0.5. Further, we also drop questions that occur in less than 4 environments because the entropy counts for low-frequency questions are not reliable.

2. **Non-existence questions**: We add `existence` questions with *'no'* as the answer for objects that are absent in a given room in the current environment, but which are

	Pixel Accuracy	Mean Pixel Accuracy	Mean IOU
single	0.780	0.246	0.163
hybrid	0.755	0.254	0.166

(a) Segmentation

	Smooth-ℓ_1
single	0.003
hybrid	0.005

(b) Depth

	Smooth-ℓ_1
single	0.003
hybrid	0.003

(c) Autoencoder

Table 2: Quantitative results for the autoencoder, depth estimation, and semantic segmentation heads of our multi-task perception network. All metrics are reported on a held out validation set.

present in the same room in other environments. For example, if the living room in the current environment does not contain a piano, but pianos are present in living rooms of other environments across the dataset, we add the question *'is there a piano in the living room?'* for the current environment with a ground truth answer *'no'*. The same is also done for `logical` questions.

3. **Object Instance Count Threshold**: We do not ask counting questions (`room_count` and `count`) when the answer is greater than 5, as they are tedious for humans.

4. **Object Distance Threshold**: We consider triplets of objects within a room consisting of an anchor object, such that the difference of distances between two object-anchor pairs is at least 2 metres. This is to avoid ambiguity in *'closer'*/*'farther'* object distance comparison questions.

5. **Collapsing Object Labels**: Object types that are visually very similar (*e.g.* *'teapot'* and *'coffee_kettle'*) or semantically hierarchical in relation (*e.g.* *'bread'* and *'food'*) introduce unwanted ambiguity. In these cases we collapse the object labels to manually selected labels (*e.g.* (*'teapot'*, *'coffee_kettle'*) → *'kettle'* and (*'bread'*, *'food'*) → *'food'*).

6. **Blacklists**:
   - **Rooms**: Some question types in the EQA dataset have room names in the question string (e.g. `color_room`, `exist`, `logical`). We do not generate such questions for rooms that have obscure or esoteric names such as *'loggia'*, *'freight elevator'*, *'aeration' etc*. or names from which the room being referred might not be immediately obvious *e.g.* *'boiler room'*, *'entryway' etc*.
   - **Objects**: For each question template, we maintain a list of objects that are not to be included in questions. These are either tiny objects or whose name descriptions are too vague e.g. *'switch'* (too small), *'decoration'* (not descriptive enough), *'glass'* (transparent), *'household appliance'* (too vague). These blacklists are manually defined based on our experiences performing these tasks.

2. CNN Training Details

The CNN comprising the visual system for our EmbodiedQA agents is trained under a multi-task pixel-to-pixel prediction framework. We have an encoder network that transforms the egocentric RGB image from the House3D renderer [1] to a fixed-size representation. We have 3 decoding heads that predict 1) original RGB values (*i.e.* an autoencoder), 2) semantic class, and 3) depth for each pixel. The information regarding semantic class and depth of every pixel is available from the renderer. The range of depth values for every pixel lies in the range $[0, 1]$ and the segmentation is done over 191 classes.

Architecture. The encoder network has 4 blocks, comprising of $\{5\times5$ Conv, BatchNorm, ReLU, 2×2 MaxPool$\}$. Each of the 3 decoder branches of our network upsample the encoder output to the spatial size of the original input image. The number of channels in the output of the decoder depends on the task head – 191, 1 and 3 for the semantic segmentation, depth and autoencoder branches respectively. We use bilinear interpolation for upsampling, and also use skip connections from the 2nd and 3rd convolutional layers.

Training Details. We use cross-entropy loss to train the segmentation branch of our hybrid network. The depth and autoencoder branches are trained using the Smooth-ℓ_1 loss. The total loss is a linear combination of the 3 losses, given by $L = L_{\text{segmentation}} + 10\times L_{\text{depth}} + 10\times L_{\text{reconstruction}}$. We use Adam [2] with a learning rate of 10^{-3} and a batch size of 20. The hybrid network is trained for a total of 5 epochs on a dataset of $100k$ RGB training images from the renderer.

Quantitative Results. Table 2 shows some quantitative results. For each of the 3 different decoding heads of our multi-task CNN, we report results on the validation set for two settings - when the network is trained for all tasks at once (hybrid) or each task independently (single). For segmentation, we report the overall pixel accuracy, mean pixel accuracy (averaged over all 191 classes) and the mean IOU (intersection over union). For depth and autoencoder, we report the Smooth-ℓ_1 on the validation set.

Qualitative Results. Some qualitative results on images from the validation set for segmentation, depth prediction and autoencoder reconstruction are shown in Figure 1.

3. Question Answering Module

The question answering module predicts the agents' beliefs over the answer given the agent's navigation. It first encodes the question with an LSTM, last five frames of the navigation each with a CNN, and then computes dot product attention over the five frames to pick the most relevant ones.

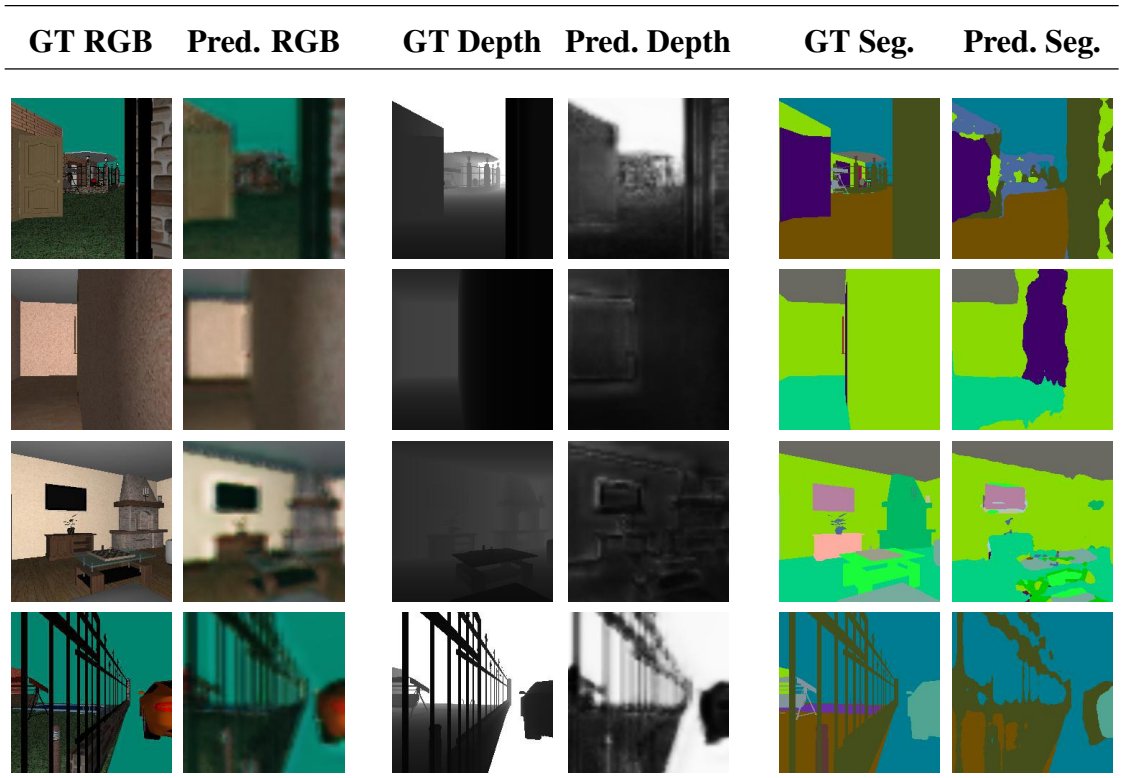


Figure 1: Some qualitative results from the hybrid CNN. Each row represents an input image. For every input RGB image, we show the reconstruction from the autoencoder head, ground truth depth, predicted depth as well as ground truth segmentation and predicted segmentation maps.

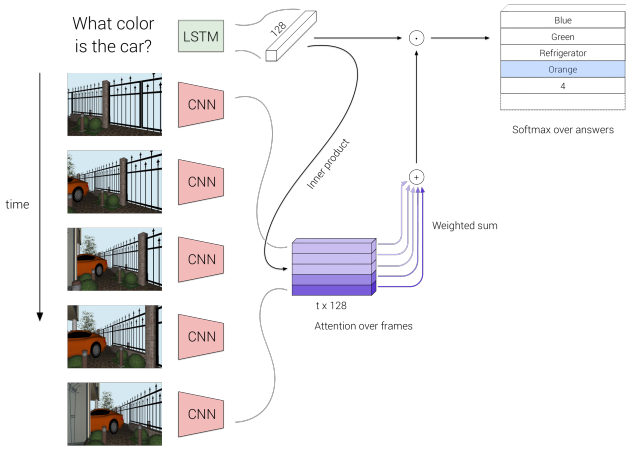


Figure 2: Conditioned on the navigation frames and question, the question answering module computes dot product attention over the last five frames, and combines attention-weighted combination of image features with question encoding to predict the answer.

Next, it combines attention-weighted sum of image features with the question encoding to predict a softmax distribution over answers. See Fig. 2.

4. Human Navigation + Machine QA

In order to contrast human and shortest-path navigations with respect to question answering, we evaluate our QA model on the last 5 frames of human navigations collected through our Amazon Mechanical Turk interface. We find the mean rank of the ground truth answer to be 3.51 for this setting (compared to 3.26 when computed from shortest-paths). We attribute this difference primarily to a mismatch between the QA system training on shortest paths and testing on human navigations. While the shortest paths typically end with the object of interest filling the majority of the view, humans tend to stop earlier as soon as the correct answer can be discerned. As such, human views tend to be more cluttered and pose a more difficult task for the QA module. Fig. 5 highlights this difference by contrasting the last 5 frames from human and shortest-path navigations across three questions and environments.

References

[1] Y. Wu, Y. Wu, G. Gkioxari, and Y. Tian, "Building generalizable agents with a realistic and rich 3D environment," *arXiv preprint arXiv:1801.02209*, 2018. 1, 3

[2] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *ICLR*, 2015. 3

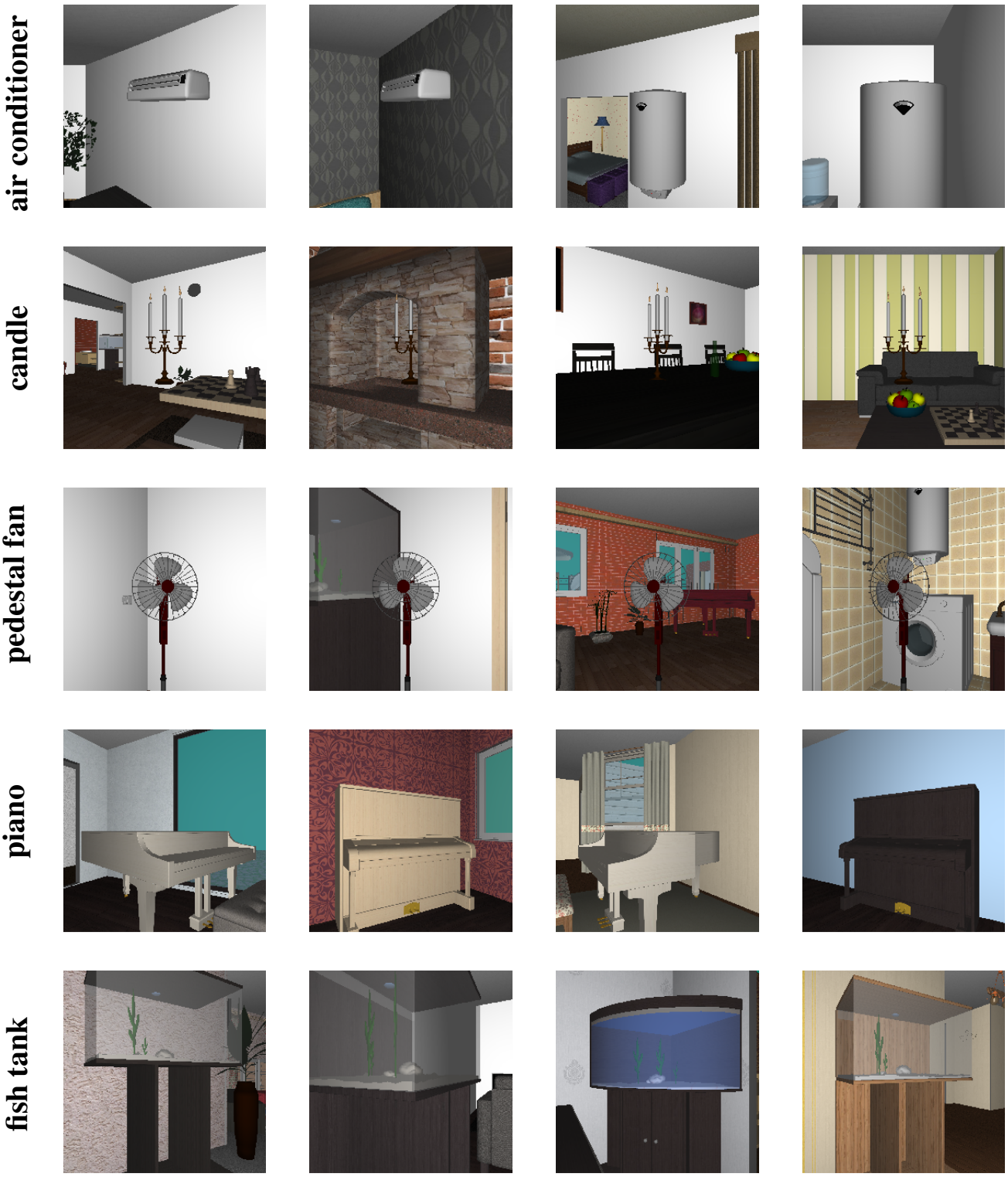
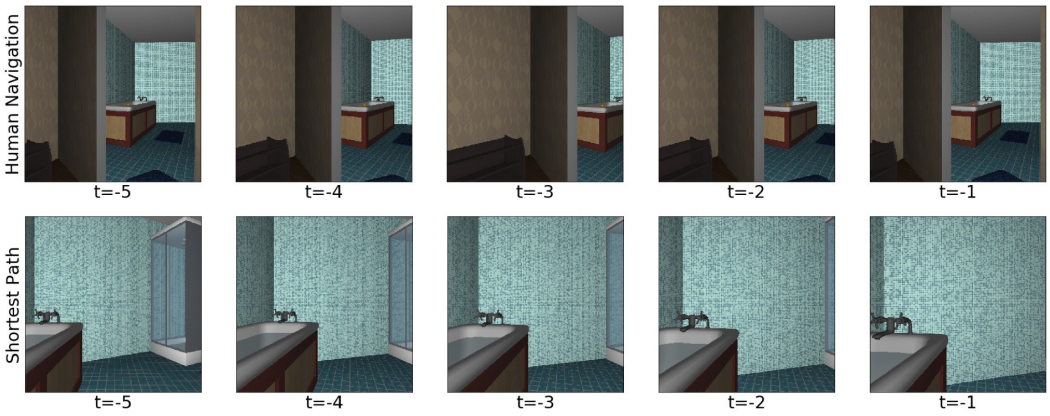
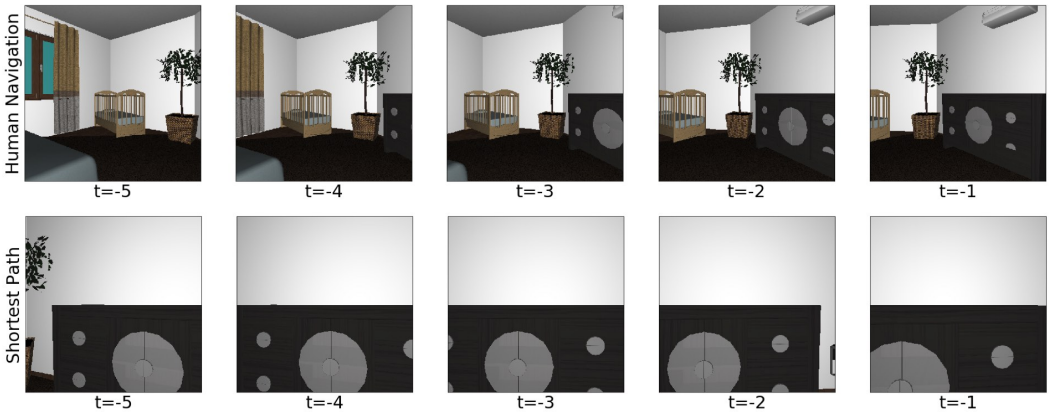


Figure 3: Visualizations of queryable objects from the House3D renderer. Notice that instances within the same class differ significantly in shape, size, and color.

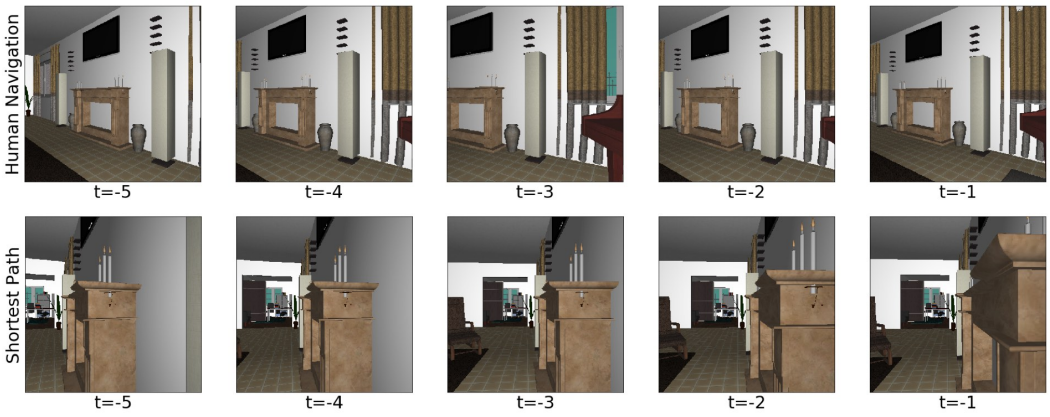
Figure 4: Visualizations of queryable rooms from the House3D renderer.



Q: What color is the bathtub?

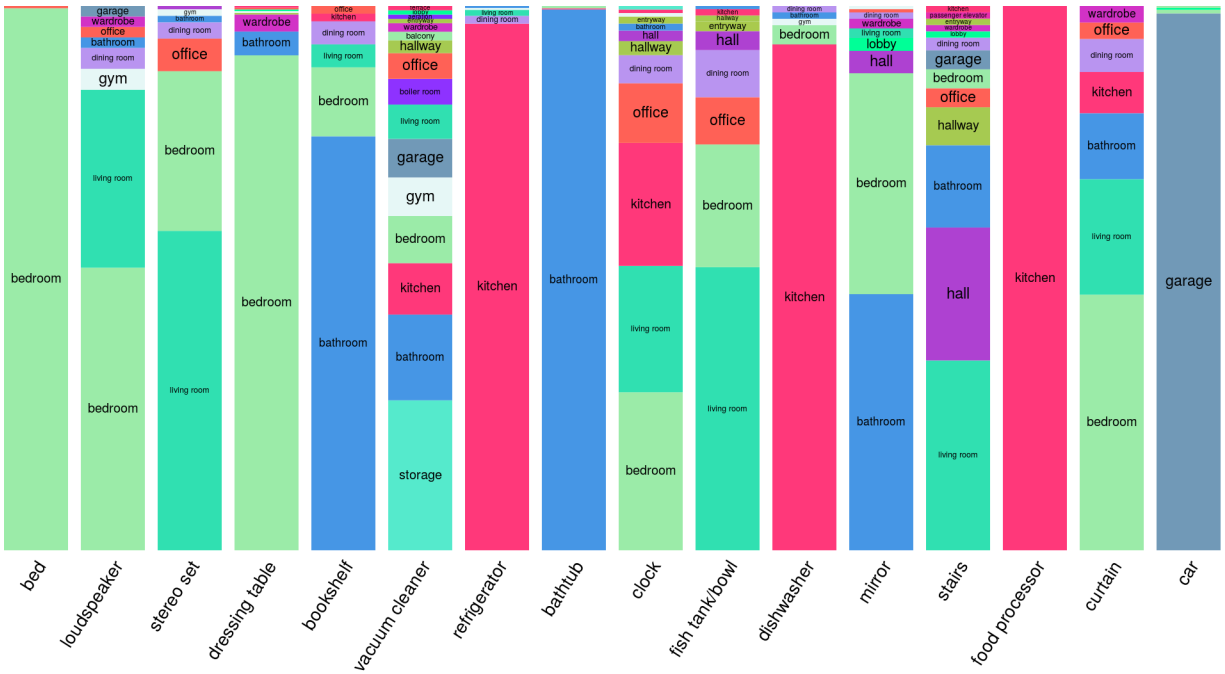


Q: What color is the dresser?

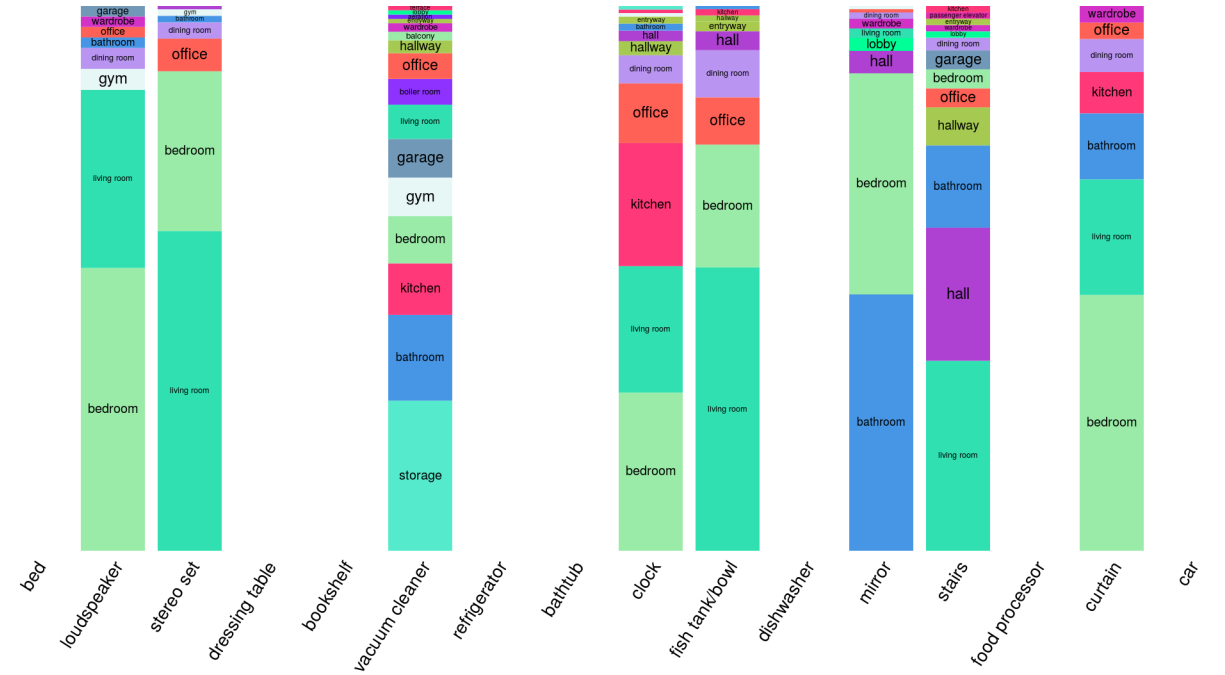


Q: What color is the fireplace in the living room?

Figure 5: Examples of last five frames from human navigation vs. shortest path.

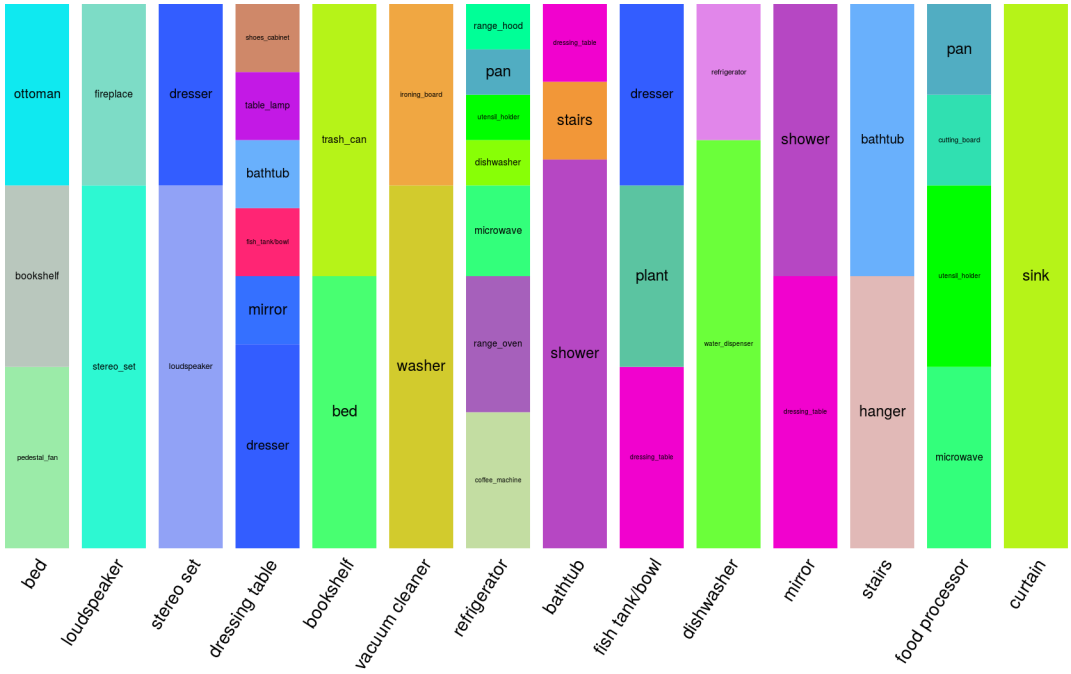


(a) `location` questions before entropy+count based filtering

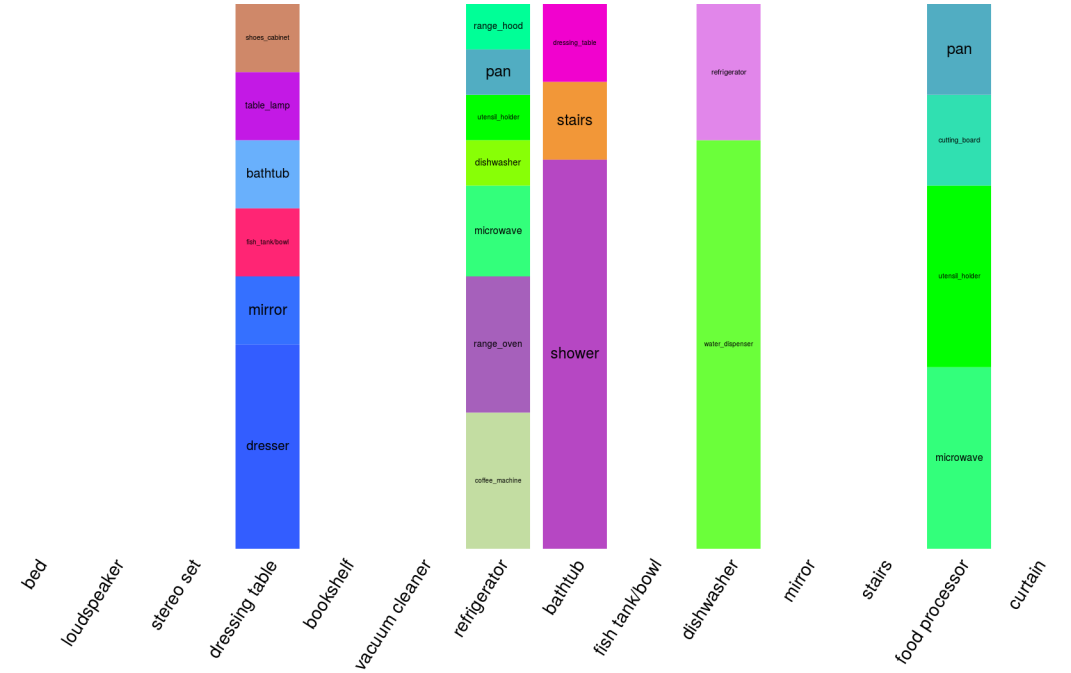


(b) `location` questions after entropy+count based filtering

Figure 6: The answer distribution for `location` template questions. Each bar represents a question of the form *'what room is the <OBJ> located in?'* and shows a distribution over the answers across different environments. The blank spaces in 6b represent the questions that get pruned out as a result of the entropy+count based filtering.

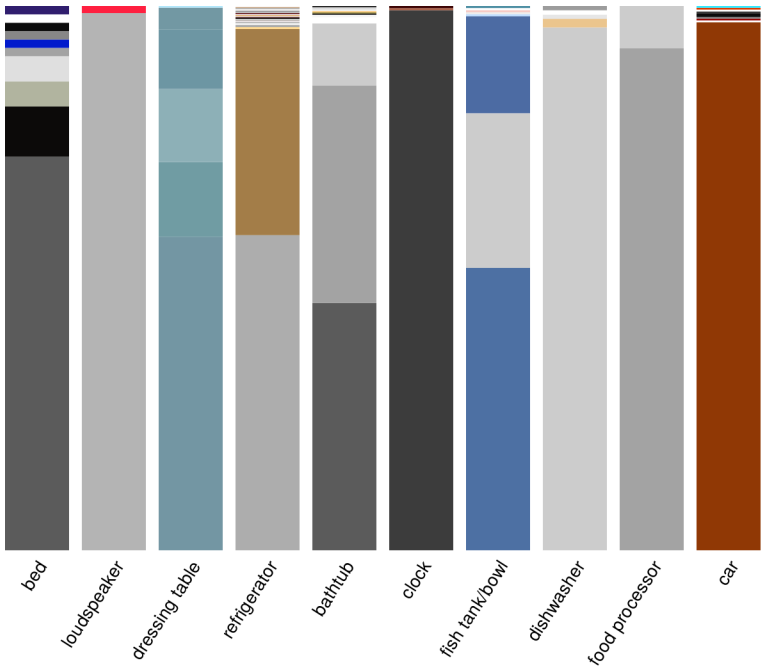


(a) `preposition` questions before entropy+count based filtering

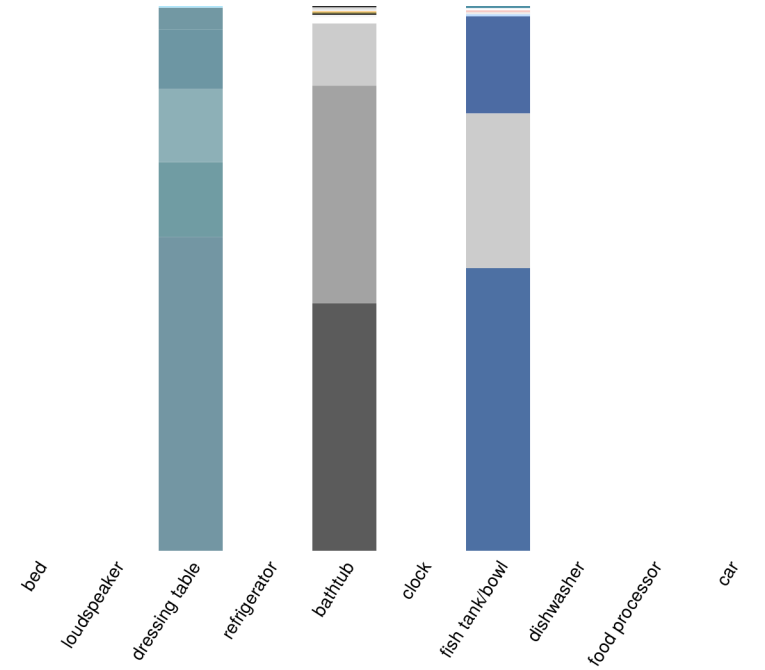


(b) `preposition` questions after entropy+count based filtering

Figure 7: The answer distribution for `preposition` template questions. Each bar represents a question of the form *'what is next to the <OBJ>?'* and shows a distribution over the answers across different environments. The blank spaces in 7b represent the questions that get pruned out as a result of the entropy+count based filtering.



(a) `color` questions before entropy+count based filtering



(b) `color` questions after entropy+count based filtering

Figure 8: The answer distribution for `color` template questions. Each bar represents a question of the form *'what color is the <OBJ>?'* and shows a distribution over the answers across different environments (the color of each section on a bar denotes the possible answers). The blank spaces in 8b represent the questions that get pruned out as a result of the entropy+count based filtering.