A Survey

In this section, we provide further information about the survey presented in Sec. 2 that guided our benchmark development. We'll discuss how we selected activities that are part of the survey, the survey design and execution, and the demographic information about the survey participants.

A.1 Activity Sources

We source activities from a combination of *time-use surveys* and WikiHow [A1]. Time-use surveys are studies that inquire about the daily time use of a population [50], making them a good proxy for daily requirements of embodied intelligence. We combine information from three time-use surveys—American [50], Harmonized European [51], and Multinational [52] Time Use Surveys—and obtain an initial set of 540 activities. The time-use surveys focus on activities that happen with enough frequency and require a significant amount of time. However, there are other activities that are essential for everyday human life, but not reflected in the time-use surveys. WikiHow articles include activities that are important for humans where they seek guidance, even if they are not as frequent as the ones included in the time-use surveys. This indicates a great potential to source additional useful and relevant daily activities. We complement the activities from the time use surveys with WikiHow article titles, of which there are 180,000+. These constitute a raw set of activities that are filtered down by feasibility, as explained in the next paragraph.

Activity filtering for feasibility: Simulation constrains which activities collected from time-use surveys and WikiHow can actually be used in BEHAVIOR-1K. We used the following criteria based on OMNIGIBSON and BDDL constraints to filter the activity pool prior to the survey:

Filtering principle	Example activity filtered out
Activity requires physics or chemistry not supported in simulation	`SteamingClothes`, `MakingSoap`
Activity involves creating or consuming media	`ReadingABook`
Activity requires more than a day in real time	`DryingSeedsOvernight`
Activity requires non-visual perceptual modalities	`SweeteningFood`
Activity requires geometric configurations too fine-grained for BDDL	`SettingUpANativityScene`
Activity is predicated on branded items	`SprayingWindex`
Activity involves other people or live animals	`AskingForARaise`

After filtering and eliminating duplicates, and including the activities from [27], 2,090 activities remain to be surveyed.

A.2 Survey Design

Our survey is structured as follows:

- **Demographic questions** requesting information about the number of people in the household, occupation, general location, and relationship between household work, automation, and livelihood (see Fig. A.1 for results).
- **Activity survey questions** requesting the value of automation of a batch of activities to the respondent. Specifically, this section comprises:
  - 50 questions, one question per activity
  - Question text: *On a scale of 1 (left) to 10 (right), rate how much you want a robot to do this activity for you.*
  - Each response uses an independent Likert score [A2] on a scale from 1 (less beneficial) to 10 (most beneficial)

We piloted alternatives for the survey about two design decisions: 1) question wording, and 2) question format. With respect to the wording for the question, we piloted three alternatives to specify the *agent*: *robot*, *assistant*, and *automation*. Our goal was to study any possible (mis)conceptions about robots that might bias the study. By way of 30 pairwise T-tests and 10 ANOVAS [A3], we did not find any statistically significant difference between the results obtained using these three wordings. With respect to the format of the question, we piloted two alternative formats: 10-point Likert, and three-element best-worst scaling. By way of Kendall's tau [A3], there was a strong correlation between the rankings determined by Likert scores and standard metrics of best-worst scaling. We, therefore, conclude that either method will give us a similar ranking, and select the more resource-efficient option (Likert).

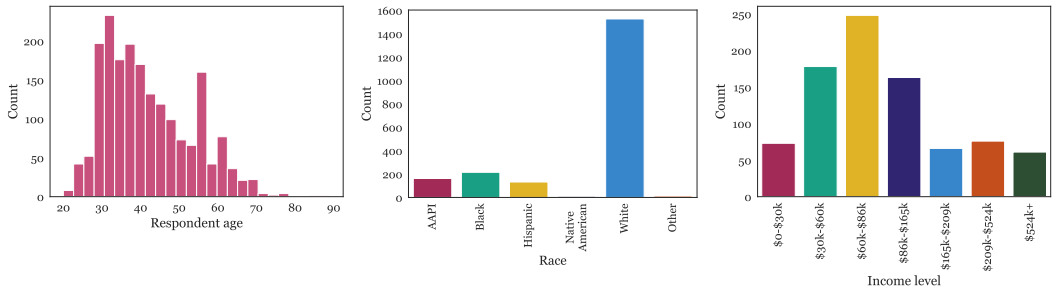


Figure A.1: **Demographic distribution of the participants in our survey.** Respondents appear to have similar demographics to the wider U.S. Mechanical Turk worker population [A5]. Not pictured: respondent gender (43.41% women, 55.50% men, 0.83% non-binary, 0.26% other), disability status (92.56% no, 5.74% yes, 1.70% prefer not to answer).

Survey collection: We deployed our survey on Amazon Mechanical Turk [A4] and collected 50 unique responses per activity. There were a total of 1,461 different respondents and their average scores ranged from 1.9 to 9.3, showing high diversity. The average score was 5.16. To ensure response quality, we repeated four questions in every survey as an attention check. If responses to a pair of repeats differed by more than two points, we considered it failed. Two or more failures led to rejection. We also rejected survey responses that had no significant variance across their responses to 50 different activities.

A.3 Demographic Information

Fig. A.1 depicts the results of our demographic questions on the participants of the survey. We observe that most adult age groups have good representation, particularly pre-retirement age groups, and are concentrated in the 30-40 group. Racially, the respondents are around 75% white, a larger proportion than the U.S. population but similar to the Mechanical Turk proportion [A5]. Other races appear to have proportions similar to but not the same as their presence on both Mechanical Turk and in the general population; this may be due to their overall small numbers in all three [A5]. There is some Native American representation, though not statistically significant. Income-wise, respondents tend to be in the $30,000-$150,000 range, particularly in the lower half.

The participants' gender is distributed by 43.41% women, 55.50% men, 0.83% non-binary, 0.26% other. Gender representation is not as even as the U.S. population but more so than the Mechanical Turk worker population [A5]. There is a small representation of non-binary individuals. When it comes to disability status, our participants reported 92.56% no-disability, 5.74% disability, and 1.70% prefer not to answer. Thus, our survey contained some but limited representation of people with disabilities. This group, together with the elderly, are groups commonly assumed to potentially benefit from robotic efforts; they could be subject to future targeted surveys.

B Activity Annotation

Obtaining the BDDL definitions for 1,000 activities requires multiple preparatory steps to gather the knowledge needed. This knowledge also becomes part of the BEHAVIOR-1K DATASET. The pipeline is outlined in Fig. A.2. We now detail each annotation other than the survey. Furthermore, quality assessment statistics for these annotations are found in Sec. B.1. We see that experienced annotators find our resulting BEHAVIOR-1K knowledge base highly accurate.

Obtaining list of objects per activity (WikiHow articles, noun phrase extraction, and manual object filtering): Our activity definition requires first defining a set of objects and properties that the annotators could use to describe an activity. We would like these domains to be natural and ecological, containing all relevant objects that humans may consider necessary for a task. To obtain such an ecologically plausible object space per activity, we parse WikiHow articles. Since WikiHow is a how-to database with a wide following and community as well as expert- and peer-review, the article texts document objects that are used and acted on in an activity. We therefore asked crowdworkers from the Upwork platform to collect articles; for robustness, we had five articles collected for each activity. We used a chunking model [A6] to extract noun phrases from the article text, then we manually filtered the noun phrases into tangible objects.

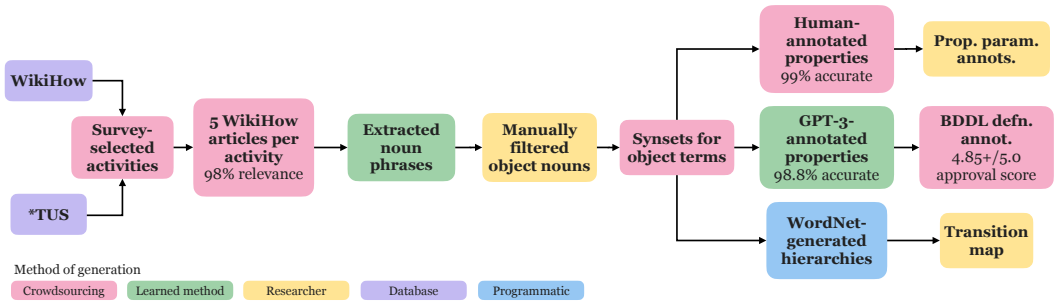


Figure A.2: **Flow chart of the annotation pipeline:** activity selection, activity article collection, object list extraction, object property and parameter annotation, BDDL annotation.

Synsets for object terms and WordNet hierarchy generation: After obtaining an object space for each activity, we take their union and crowdworkers match each one to a WordNet [57] synset. This eliminates word-sense ambiguity in the knowledge base and creates a hierarchical structure in the overall object space via the WordNet hierarchy. This process yielded 1,484 total synsets.

Property annotations: Each object that is a leaf-level synset of the WordNet hierarchy is associated with the set of object properties in BEHAVIOR-1K, each of which is fully simulatable in OMNIGIBSON (full list of properties in Table A.1). The properties define which predicates can apply to the object - e.g. an object having the `cookable` property means it can be `cooked` and `not cooked`. This association is done task-agnostically because objects/states that are irrelevant or undesirable for a specific activity will still provide important learning signals. All properties that apply to many of the 1,484 synsets and therefore require a large-scale annotation are done by either GPT-3 [58] or crowdworkers. GPT-3 is used for all properties (total of nine) for which the Hamming distance and false-positive rate for a sample compared to a human-annotated ground-truth are both less than 10%. Human annotation is used for five more properties. The remaining properties either apply to very few objects and are therefore annotated manually, or in most cases can be inferred from the other annotations (e.g. everything "cookable" is "overcookable") and are determined analytically from the other property annotations.

This annotation is done only for leaf-level synsets, then properties of higher-level synsets are inferred from the leaf-level annotations. This prevents definition unsolveability. In more detail: as shown in Fig. A.2, the hierarchical structure is applied to the object space. This allows activity definition annotators to refer to them instead of leaf-level synsets; for example, a `PuttingAwayGroceries` definition can have "five `edible_fruit.n.01`s" rather than "two `apple.n.01`s and three `banana.n.01`s", allowing more variation in activity instantiation because far more object models are valid. However, 3-D object models are generally only attached to leaf- or near-leaf-level synsets, meaning that a definition may have a higher-level synset with various predicates attached to it, and at simulation time the object model (associated with one of the synset's descendants) must have those states simulated. A problem might occur when e.g. `container.n.01` is annotated as `fillable` and the definition calls for a `container.n.01` to be `filled` with a `liquid`, but the actual model that gets sampled into the simulator to satisfy `container.n.01` is a cloth bag that doesn't support being `filled` with a `liquid`. So to avoid this unsolveability, the properties of any non-leaf synset are exactly the *intersection of all its descendants' properties*.

Object property	Annotation method	Prompt to annotator	Example objects
assembleable	manual	N/A	desk.n.01, table.n.02
boilable	prog. (all liquid)	N/A	champagne.n.01, beef_broth.n.01
breakable	human	Mark if the object can be broken into smaller pieces by a human dropping it on the floor without a tool.	wine_bottle.n.01, room_light.n.01
cleaningTool	GPT-3	Is a [object] designed to clean things?	scrub_brush.n.01, pipe_cleaner.n.01
cloth	manual	N/A	hammock.n.02, canvas.n.01
coldSource	GPT-3	Is [object] a source of cold?	refrigerator.n.01, ice.n.01
cookable	GPT-3	Can a [object] be cooked?	biscuit.n.01, pizza.n.01
deformable	prog. (all softBody, cloth, rope)	N/A	tortilla.n.01, clay.n.01
fireSource	GPT-3	Is a [object] designed to create fire?	lighter.n.01, sparkler.n.01
flammable	human	Mark if the object can catch fire (i.e. burn with a flame).	candle.n.01, mail.n.01

foldable	prog. (all cloth, softBody)	N/A	tortilla.n.01, jean_jacket.n.01
freezable	prog. (all heatable)	N/A	olive_oil.n.01, ginger_beer.n.01
heatable	prog. (all objects)	N/A	oil.n.01, fish_knife.n.01
heatSource	GPT-3	Is a [object] a source of heat?	oven.n.01, toaster.n.01
liquid	GPT-3	Is a [object] a liquid?	gasoline.n.01, liquid_soap.n.01
meltable	manual	N/A	cheese.n.01, chocolate.n.02
openable	human	Mark if the object is designed to be opened.	mixer.n.04, keg.n.01
overcookable	prog. (all cookable)	N/A	gazpacho.n.01, jam.n.01
physicalSubstance	manual	N/A	flour.n.01, salt.n.02
rope	manual	N/A	ribbon.n.01, fairy_light.n.01
sliceable	GPT-3	Can a [object] be sliced easily by a human with a knife?	sweet_corn.n.01, sandwich.n.01
slicingTool	GPT-3	Can a [object] slice an apple?	blade.n.09, razor.n.01
soakable	GPT-3	Can a [object] absorb liquid?	sponge.n.01, tea_bag.n.01
softBody	manual	N/A	dough.n.01, pillow.n.01
substance	prog. (all liquid, vis.Subst., phys.Subst.)	N/A	water.n.06, milk.n.01
toggleable	human	The object can be switched between a finite number of *discrete* states and is designed to do so.	hot_tub.n.01, light_bulb.n.01
visualSubstance	manual	N/A	coriander.n.02, cocoa_powder.n.01
waterSource	manual	N/A	sink.n.01
wetable	prog. (all non-substance	N/A	lasagna.n.01, ashtray.n.01
unfoldable	prog. (all foldable)	N/A	tissue.n.02, foil.n.01

Table A.1: Properties annotated for BEHAVIOR-1K for each object category and included in the BEHAVIOR-1K DATASET knowledge base

Property parameter annotations: BDDL separates objects and object states (i.e. terms and predicates), but simulating realistic "cooking" or "filling" requires information for object-object property tuples, not just objects or object properties alone. For example, in the real world, an apple.n.01 and a chicken.n.01 do not become cooked at the same temperature, so simply knowing they are both cookable is insufficient. BEHAVIOR-1K DATASET therefore includes manual annotation of several property parameters that are (object, object property) specific, such as cookTemperature for all cookable objects.

Examples of object-property pairs and parameters:

- Temperature required for cookable object to go from (cooked object) = False to (cooked object) = True
  - crab.n.05: must reach **63°C**
  - squash.n.02: must reach **58°C**
  - meatball.n.01: must reach **63°C**
  - chicken_leg.n.01: must reach **74°C**
- Temperature generated by heatSource object. For toggleable heatSources, this may require toggledOn(object) = True
  - toaster_oven.n.01: generates **204°C**
  - ember.n.01: generates **1093°C**
  - hand_blower.n.01: generates **45°C**
  - coffee_maker.n.01: generates **93°C**

Transition rules: There are many activities that involve complex chemical and physical processes that are beyond the capability of the state-of-the-art simulation technology. For example, blending different fruits into a smoothie or sanding a rusted surface are extremely difficult to simulate, but the agent actions to complete these tasks are still within reach, e.g. placing the fruits inside a blender. Therefore, in ordet to support these type of activities, we create a set of transition rules that will be used by OMNIGIBSON to bypass the underlying physics but still produce visually realistic physical transitions, e.g. smoothie particles being generated inside the blender after the blender is turned on.

Examples of transition rules:

- Composition and decomposition of objects
  - Transition rule used in make a strawberry slushie
    - **Inputs:** 4+ strawberry.n.01s, ice.n.01, lemon_juice.n.01, agave.n.01

        - **Transition machine:** blender.n.01
        - **Outputs:** smoothie.n.01
    - Transition rule used in make gazpacho
        - **Inputs:** basil.n.03, salt.n.02, black_pepper.n.02, tomato_juice.n.01, cucumber.n.02, water.n.06, lemon_juice.n.01
        - **Transition machine:** saucepan.n.01
        - **Outputs:** gazpacho.n.01
- Realistic cleaning rules
    - Transition rule used in CleanTheExteriorOfYourGarage
        - **Substance covering object:** paint.n.01 or spray_paint.n.01
        - **Objects needed to remove:** cleaningTool and (solvent.n.01 or acetone.n.01)
    - Transition rule used in CleanYourRustyGardenTools
        - **Substance covering object:** rust.n.01 or patina.n.01 or incision.n.01 (simulated as particles but exposed to annotators as unary scratched predicate) or tarnish.n.01 (simulated as particles but exposed to annotators as unary tarnished predicate)
        - **Objects needed to remove:** emery_paper.n.01 or whetstone.n.01

Activity definitions in BDDL: Finally, the object spaces and the relevant properties and predicates they enable are offered to lay annotators to generate BDDL definitions. Annotators build activity definitions using an annotation interface that includes a visual version of BDDL [27] (details on new features in BDDL are in Sec. C). The annotation interface enforces the requirements needed to make definitions logically solvable and well-formed. We collect one definition per activity for a total of 1,000 BDDL activity definitions.

Listings 1, 2, and 3 show examples of BEHAVIOR-1K activity definitions, while Listing 4 and 5 show examples of BEHAVIOR-100 definitions. We see that while the numbers of objects and literals are similar, speaking to the fact that both benchmarks have reached similar scale and detail for the activities they have, BEHAVIOR-1K has far larger and more detailed activity distribution. The MakeScones activity involves transition rules that turn ingredients in :init to scones in :goal, a process that will require the listed mixer and oven in OMNIGIBSON. By contrast, cooking in BEHAVIOR-100 was limited to single objects transitioning from not cooked to cooked; other benchmarks are similar or lack cooking entirely. CleanYourLaundryRoom involves specific cleansing agents to clean mold, whereas BEHAVIOR-100 cleaning tasks only had two types of "dirtiness" (stained and dusty) and the only rule was to use water in the case of stained. FixingMailbox also shows rust-specific cleaning (requiring emery paper) and transitioning from the broken to not broken states via transition rule-based simulation.

B.1 Quality assessment of BEHAVIOR-1K annotations

	Article Collection	Object Extraction	Human Properties	GPT-3 / Machine Properties
Accuracy (Approval Rate)	0.974	0.968	0.990	0.988
F1-score	0.984	0.990	0.912	0.930
False Discovery Rate	0.026	0.032	0.031	0.029
False Positive Rate	N/A	N/A	0.002	0.003

Table A.2: Quality check results for object and object property annotations: for each stage in the annotation pipeline, report human verification results. The F1-score is $\frac{2}{\text{precision}^{-1}+\text{recall}^{-1}}$. The False Discovery Rate is FDR = FP/(TP+FP) and the False Positive Rate FPR = FP/(TN+FP). The high accuracy and F1-scores and low FDRs and FPRs show that our annotation results are of high quality.

	Activity Definition			
	Question 1	Question 2	Question 3	Question 4
Average Rating	4.875	4.942	4.967	4.975
Standard Deviation	0.331	0.234	0.364	0.156

Table A.3: Quality check results for activity definitions. Q1: Are the objects listed relevant to the definition of the task?, Q2: Do the initial placements/locations of objects make sense?, Q3: Are the actions to perform ("goals") relevant to the definition of the activity?, Q4: Is this definition reasonable?
Each activity definition was evaluated on a scale of 1-5 for each of the questions, and the average score and standard deviation are presented. The increased granularity still reflects the high quality of our results.

Our quality control investigation shows us that all labeling done by crowdworkers and GPT-3 is of the highest quality. Five crowdworkers with an extensive background in data labeling for large

machine learning projects, coding, or data verification affirmed the results from earlier crowdworkers. The accuracies for all the labeling tasks were all above 96%, the F1 scores were above 91% and the false positive and false discovery rates were between 2-3% as shown in Table A.2. We noticed that the Synset Verification is a bit lower in accuracy than the other labeling tasks. This may be due to subtleties in acceptable synset definitions: for example, a "reasonably narrow hypernym" of a word not found in WordNet [57] is acceptable, such as a "dispenser" for a "soap dispenser", but there may be gray areas regarding what is considered "reasonably narrow" (e.g. would a "hand tool" be reasonable as a replacement for a "soap dispenser"?)

The activity definition process was evaluated with more granularity using a Likert scale and an array of questions. We found that the response values had consistently high averages (very close to the maximum score of 5) and low standard deviations as shown in Table A.2. We also found no significant discrepancy for activity definition scores across topics (e.g. tasks related to "cleaning"), showing that the BDDL definitions were rated uniformly across the span of activity categories.

C New BDDL features

BEHAVIOR-1K uses an expanded version of the BDDL used in BEHAVIOR-100 [27], in order to support the new set of diverse activities. There are four new features here: 1) representation of substances, 2) three-valued predicates, 3) composition and decomposition of objects, and 4) variable-arity predicates. We now detail the need for each of these, why the BDDL from BEHAVIOR-100 cannot support them, and the design in this version of BDDL.

Representation of substances: BEHAVIOR-1K introduces *substances*, objects that are arbitrarily subdivideable and do not obey clear instance boundaries such as `water.n.06` or `flour.n.01`. The lack of instance boundaries is difficult with traditional PDDL/BDDL: for example, if there are two bottles of orange juice where the juice inside one is called `orange_juice.n.01_1` and the juice inside the other is called `orange_juice.n.01_2`, any mixing of the particles makes it near-impossible for the agent to satisfy a `:goal` condition pertaining to an instance. Even with quantification, a condition like `exists (orange_juice.n.01) (filled orange_juice.n.01 glass.n.01_3)` is still unfair: if the agent mixes particles from the two different `orange_juice` instances such that together they `filled` `glass.n.01_3` but neither instance alone has enough particles in `glass.n.01_3` to fill it, the `:goal` will not be met.

We simply enforce that there is up to one instance of any `substance` in a definition. The annotator can still control quantity by spawning it in as many containers as desired. Unlike labeling every particle separately, this maintains a compact representation. This does not allow for some of the substances to be different from some (e.g. some of the orange juice is cold and some is hot), but we consider this an acceptable limitation.

Furthermore, particle-based objects can be computationally expensive to simulate. When an annotator uses `orange_juice.n.01` only as a container of orange juice and not actually the particles (e.g. in a `PuttingAwayGroceries` activity), this becomes a waste of computational resources. Therefore, we introduce a custom container synset for every substance that has the same properties and WordNet hierarchy structure as `bottle.n.01` and instruct annotators to use it if all they want is the container.

Three-valued predicates: A common issue in BEHAVIOR-100 activities is that success on predicates involving naturally continuous-valued quantities is sudden and arbitrary: cracking a window a little bit suddenly changes it from `not open` to `open`, and the activity from undone to done – even though the window is not a typical human conception of "open". PDDL 2.1 [A7] has numerical fluents and derived predicates, which offer continuous states. However, this granularity may not be crucial to the high-level activities in BEHAVIOR-1K, and the concept is also difficult to communicate to lay annotators.

We therefore treat certain predicates as three-valued by having two Boolean predicates where the negations are the same, such as `filled` and `empty` (rather than just `not filled`). The annotators still see one Boolean predicate, in this case `filled`, and negate it to say the opposite, but we assume that when they negate, they mean something decisively empty. This is a strong assumption but generally safe as people tend not to add expressions that seem like common sense. Wherever we see the predicate negated in the definition, we switch it out with the other predicate. This applies to `filled`/`empty`, `open`/`closed`, and `folded`/`unfolded`.

To ensure that the definition is logically the same after the switch, the underlying BDDL implementation converts the definition using De Morgan's Law such that all negations are only applied to atomic formulae before the switch occurs.

Composition and decomposition of objects: In standard PDDL/ BDDL, the objects in `:objects` are assumed to persist throughout the activity. There is no concept of creating a new object that did not appear in the `:init` or explicitly destroying an object.

To enable our transition rules that involve turning some objects into others, we require a representation that will provide the desired information unambiguously. The `:objects` section cannot be inferred simply from a `:goal` that has new objects in it, because `:goal` is not exhaustive. We therefore introduce the `future` predicate, used in `:init` on all objects that do not appear in the scene when the agent enters, but must be present for the `:goal` to be satisfied. All objects in `future` predicates appear in `:objects` and they cannot appear in any other literals in `:init`. This approach takes inspiration from [A8], which updates a reference to an object (e.g. a wall) as it keeps changing form as more sub-objects (e.g. blocks) are added to it.

Variable-arity predicates: Certain predicates can take various numbers of objects while indicating the same underlying concept. For example, in BEHAVIOR-1K, the `mixed` predicate can take any number of objects: the expressions `(mixed apple.n.01_1 apple.n.01_2 peach.n.03_1)` and `(mixed apple.n.01_1 apple.n.01_2 peach.n.03_1 banana.n.01_1)` give different numbers of objects to `mixed`, but mean the same thing and are checked the same way in OMNIGIBSON.

Logic predicates are fixed-arity by default, but work exists to expand to variable-arity predicates [A9]. PDDL uses this assumption to specify predicates in its domain that can be used with PDDL actions to create plans. However, because BDDL is used for benchmarking, it is process-agnostic and does not assume any planning method [27]. Therefore, a domain file does not have any effect in BDDL, so a variable-arity predicate can be specified (rather than potentially infinite fixed-arity versions) to maintain compactness. This requires no structural changes to BDDL.

D Scene and Object Models

In this section, we provide more details about the object and scene models presented in BEHAVIOR-1K DATASET, including the selection, modeling, and annotation processes.

The survey outlined in Sec. 2 provided us with a list of 1000 activities that humans prefer robots to perform. However, these activities are not restricted to a unique type of scene (e.g., houses). We chose scene types necessary to cover the BEHAVIOR-1K activities, including eight scene types: houses (15), houses with gardens (8), hotels (3), offices (5), grocery stores (4), generic halls (4), restaurants (6), and schools (5) (see Table A.4). These scenes cover activities that require a specific type (e.g., shopping, cooking, restocking) and activities that are generic and could happen in multiple scene types (e.g., cleaning). We also annotated how many activities could be performed on each type to guide us on how many instances of each scene type should we include in BEHAVIOR-1K DATASET. The main type of scene is still households: we improved 15 household scenes from BEHAVIOR-100 and annotated it further with light sources, new textures, etc. We then acquired additional instances of each scene type from online marketplaces such as TurboSquid [A10].

Several of the available scene models in the marketplaces did not contain all the necessary rooms to perform the natural activities in BEHAVIOR-1K, e.g., restaurants did not include the kitchen, offices did not include the restrooms, or houses did not include the gardens. We collected separated models for those and contracted professional 3D designers to connect them.

Scene models were acquired with the necessary object models. However, they were not enough to cover the objects required by the activities in BEHAVIOR-1K. We acquired additional models to support the activities, as provided by the activity annotation process described in Sec. B, totaling 5,000+ object models from 1,200+ categories. The diversity of the object categories included in the dataset can be observed in Fig. A.7.

As provided by the 3D vendors, the scene and object models cannot be used directly for the simulation of the 1000 activities in BEHAVIOR-1K in OMNIGIBSON due to 1) lack of category annotation, 2) lack (or incorrect) of light sources in scenes, 3) incorrect part segmentation, 4) lack of articulation, 5) missing interactive elements like buttons, 6) lack of a unified canonical frame orientation for sampling, and 7) poor physical properties for realistic simulation. We manually cleaned and annotated all scenes and objects to correct these elements.

We will publicly release all the scenes and models to be used by other researchers. The models will be encrypted and only be used within OMNIGIBSON in order to comply with the rights of the model authors and the vendors' agreement. The documentation of the annotation process as well as source code for the pipeline will similarly be released on our website, allowing users to easily import their own objects and scenes into OMNIGIBSON for use in BEHAVIOR-1K activities.

Scene Type	Scene Name	Obj. Cnt.	Syn. Cnt.	Rm. Cnt.	Room Types	Example Activities
BEHAVIOR-100 Houses	Beechwood_0_int	136	32	8	bathroom, corridor, dining_room, entryway, kitchen, living_room, private_office, utility_room	clean a hot water dispenser, freeze lasagna, clean batting gloves
	Beechwood_1_int	129	21	9	bathroom, bedroom, childs_room, closet, corridor, playroom, television_room	clean your kitty litter box, store baby clothes, cleaning pet bed
	Benevolence_0_int	21	10	4	bathroom, corridor, empty_room, entryway	laying clothes out, pack a pencil case, sweeping outside entrance
	Benevolence_1_int	74	22	5	corridor, dining_room, kitchen, living_room, storage_room	hanging blinds, sorting volunteer materials, wash baby bottles
	Benevolence_2_int	63	23	5	bathroom, bedroom, corridor	clean walls, washing fabrics, folding clean laundry
	Ihlen_0_int	68	21	7	bathroom, corridor, dining_room, garage, living_room, storage_room	de-clutter your garage, sorting household items, set up a home office in your garage
	Ihlen_1_int	147	27	9	bathroom, bedroom, corridor, dining_room, kitchen, living_room, staircase	clean jewels, clean green beans, hanging up curtains
	Merom_0_int	82	27	6	bathroom, childs_room, living_room, playroom, storage_room, utility_room	changing dog's water, wash a bra, wash a wool coat
	Merom_1_int	123	28	8	bathroom, bedroom, childs_room, corridor, dining_room, kitchen, living_room, staircase	store an uncooked turkey, drying table, clean flip flops
	Pomaria_0_int	71	22	6	bathroom, bedroom, corridor, private_office, television_room	prepare sea salt soak, clean sheets, dispose of medication
	Pomaria_1_int	89	24	6	bathroom, corridor, kitchen, living_room, pantry_room, utility_room	store brownies, clean a book, baking sugar cookies
	Pomaria_2_int	42	18	3	bathroom, bedroom, corridor	replacing screens, clean baby toys, putting up posters
	Rs_int	72	32	5	bathroom, bedroom, entryway, kitchen, living_room	dispose of a pizza box, putting food in fridge, putting out condiments
	Wainscott_0_int	187	35	9	bathroom, bedroom, dining_room, kitchen, living_room	make microwave popcorn, make frozen lemonade, make cake mix
	Wainscott_1_int	150	26	10	bathroom, bedroom, corridor, exercise_room, playroom, private_office, utility_room	decorating for religious ceremony, stash snacks in your room, disinfect laundry
BEHAVIOR-100 Houses+Garden	Beechwood_0_garden	335	45	9	bathroom, corridor, dining_room, entryway, garden, kitchen, living_room, private_office, utility_room	tidy your garden, opening doors, wash goalkeeper gloves
	Merom_0_garden	197	51	8	bathroom, childs_room, garden, living_room, playroom, sauna, storage_room, utility_room	clean your kitty litter box, clean a glass windshield, hang icicle lights
	Pomaria_0_garden	260	49	7	bathroom, bedroom, corridor, garden, private_office, television_room	putting up Christmas lights outside, prepare a hanging basket, clean a patio
	Rs_garden	185	47	6	bathroom, bedroom, entryway, garden, kitchen, living_room	cleaning driveway, clean a longboard, roast nuts
	Wainscott_0_garden	712	61	10	bathroom, bedroom, dining_room, garden, kitchen, living_room	clean an espresso machine, clearing food from table into fridge, painting porch
New Houses	house_single_floor	1375	137	23	bathroom, bedroom, childs_room, closet, corridor, dining_room, empty_room, entryway, garden, kitchen, living_room, sauna, utility_room	turning out all lights before sleep, iron curtains, packing hobby equipment
	house_double_floor_lower	304	79	6	bathroom, corridor, garage, garden, kitchen, living_room	wash towels, clean a fish, clean brass
	house_double_floor_upper	325	78	5	bathroom, bedroom, television_room	clean a saxophone, taking down curtains, clean walls
Grocery Stores	grocery_store_asian	3402	79	2	bathroom, grocery_store	buy alcohol, buy boxes for packing, buy a microwave oven
	grocery_store_cafe	6994	71	4	bar , bathroom, dining_room, grocery_store	defrost meat, clean reusable shopping bags, buy a good avocado
	grocery_store_convenience	1889	82	2	bathroom, grocery_store	washing windows, picking up prescriptions, buy food for vegetarians

	grocery_store_half_stocked	3804	51	2	bathroom, grocery_store	buying gardening supplies, buy basic garden tools, buy boxes for packing
Halls	hall_arch_wood	78	11	2	bathroom, empty_room	clean walls
	hall_train_station	141	15	2	bathroom, empty_room	clean cement
	hall_glass_ceiling	116	20	2	bathroom, empty_room	preparing food for a fundraiser
	hall_conference_large	3372	26	2	bathroom, conference_hall	distributing event T-shirts
Hotels	hotel_gym_spa	322	43	9	bathroom, corridor, gym , hammam , locker_room , sauna, spa	turning on the hot tub, adding chemicals to hot tub, clean a sauna
	hotel_suite_large	218	51	2	bathroom, bedroom	clean vinyl shutters, cleaning computer, clean white marble
	hotel_suite_small	48	28	2	bathroom, bedroom	clean cork mats, putting out clean towels, clean a shower
Offices	office_bike	479	49	3	bathroom, break_room, shared_office	set up a webcam, set up two computer monitors, clean an office chair
	office_cubicles_left	787	44	12	bathroom, conference_hall, copy_room, corridor, lobby, meeting_room, private_office, shared_office	clean up your desk, clean a LED screen, laying out snacks at work
	office_cubicles_right	406	37	11	bathroom, conference_hall, copy_room, corridor, lobby, meeting_room, private_office, shared_office	making photocopies, clean a LED screen, clean a keyboard
	office_large	1151	46	24	bathroom, break_room, conference_hall, copy_room, corridor, lobby, meeting_room, phone_room, private_office, shared_office	emptying trash cans, putting meal in fridge at work, brewing coffee
	office_vendor_machine	225	45	4	bathroom, break_room, meeting_room, shared_office	clean a computer monitor, make the workplace exciting, dispose of glass
Restaurants	restaurant_asian	1221	56	3	bathroom, dining_room, kitchen	grill vegetables, cook tofu, clean clams
	restaurant_brunch	1096	70	4	bar , bathroom, dining_room, kitchen	stock a bar, cook squid, washing vegetables
	restaurant_cafeteria	292	45	3	bathroom, dining_room, kitchen	clean a popcorn machine, store coffee beans or ground coffee, roast meat
	restaurant_diner	174	39	3	bar , bathroom, dining_room	sweeping floors, installing smoke detectors, clean a lobster
	restaurant_hotel	1080	81	4	bathroom, dining_room, kitchen, lobby	clean eggs, setting table for coffee, clean a pizza stone
	restaurant_urban	1368	90	4	bar , bathroom, dining_room, kitchen	cook pumpkin seeds, putting roast in oven, thaw frozen fish
Schools	school_biology	717	53	4	bathroom, biology_lab , corridor	clean an eraser, turning sprinkler off, clean a glass pipe
	school_chemistry	890	69	4	bathroom, chemistry_lab , corridor	clean clear plastic, clean an eraser, clean a whiteboard
	school_comp_lab_infirmary	828	61	6	bathroom, computer_lab, corridor, infirmary	clean a computer monitor, prepare an emergency school kit, clean a keyboard
	school_geography	556	42	5	bathroom, classroom, corridor	clean gold, clean a glass pipe, clean an eraser
	school_gym	853	36	7	bathroom, corridor, gym , locker_room	clean a dirty baseball, dispose of glass, wash towels

Table A.4: Statistics and information for each of 50 scenes in B1K: scene type, name, number of objects, unique synsets and rooms, room types, and example activities

E Simulation

E.1 Extended Object States and Logical Predicates in OMNIGIBSON

OMNIGIBSON extends the infrastructure of extended object states and logical predicates from iGibson 2.0 [59] to accommodate the diversity and realism of BEHAVIOR-1K.

E.1.1 Extended object states associated with object category properties

For computational efficiency purposes, not all extended object states need to be maintained and updated for every object. For example, `cookable` objects like apples need to keep track of their `Temperature`, but probably tables don't need to, at least for the purpose of simulating BEHAVIOR-1K activities. Hence, given the object category property annotation from BEHAVIOR-1K DATASET (see Table A.1), OMNIGIBSON selectively keeps track of a subset of the extended states for objects

of each category. The mapping from the object category property to the required object states can be found in Table A.5.

Object category property	Required extended object states
cookable	MaxTemperature, Temperature
overcookable	MaxTemperature, Temperature
freezable	Temperature
flammable	Temperature
heatable	Temperature
metable	Temperature
soakable	SoakedLevel
toggleable	ToggledState
sliceable	SlicedState
breakable	BrokenState
heatSource	ToggledState
fireSource	ToggledState
coldSource	ToggledState
waterSource	ToggledState

Table A.5: Extended object states associated with object category properties. Adapted from [59].

E.1.2 Object model properties

We also need to perform physical and semantic annotation for each object model so that they can be realistically simulated and support the evolution of extended object states. Some of the physical properties can be programmatically generated from the 3D assets, such as `Shape`, `KinematicStructure` and `StableOrientations`, while others need to be annotated (included in BEHAVIOR-1K DATASET), such as `Weight`. `Type` of an object model, which is also derived from the object category property annotation from BEHAVIOR-1K DATASET (see Table A.1), determines how the object will be simulated in OMNIGIBSON. For example, cloths and fluids will be simulated using underlying particle systems. Also, some of the object category property requires additional semantic annotation for each model of that category, e.g. for `waterSource` objects, `WaterSourceLink` is required to indicate the exact location to generate water. An exhaustive list of all the object model properties can be found in Table A.6.

E.1.3 Object states

While the object properties mentioned above stay the same during simulation (i.e. an apple is always `cookable` and a sink always have the same `WaterSourceLink` defined in its local frame), the object states could change. The underlying physics engine handles the kinematic state changes, such as `Pose`, `AABB`, `JointStates`, `ParticlePositions` for fluids and cloths, etc. On top of these, OMNIGIBSON handles the non-kinematic state changes that are essential for logical predicates (described in the next subsection). For example, OMNIGIBSON update the `Temperature` of objects at every simulation step by checking if they are near/inside any `heatSource` or `coldSource`. An exhaustive list of all the object states can be found in Table A.7.

E.1.4 Logical predicates as checking functions

For each logical predicate that is relevant for BEHAVIOR-1K activities, we define a checking function that maps a given physical state (kinematic and non-kinematic) into a binary logical state that BDDL operates on. For example, `OnTopOf` is based on the pose of two objects and their contact information whereas `Cooked` is based on the `Temperature`. The details of the checking functions for all the logical predicates can be found in Table A.8.

E.1.5 Logical predicates as sampling functions

For each logical predicate that is relevant for BEHAVIOR-1K activities, we also define a generative function that samples a valid physical state given a binary logical state. This functionality is essential for infinite activity initialization. For example, if the initial conditions of the activity require a plate to be `OnTopOf` a dining table, there are an infinite number of exact poses of the plate that can satisfy this condition. Our generative functions will find a valid solution and physically put the plate on top

Object model property	Relevant object category property	Description
Shape		Model of the 3D shape of each link of the object
Weight		Weight of the object
CenterOfMass		Mean position of the matter in the object
MomentOfInertia		Resistance of the object to change its angular velocity
KinematicStructure		Structure of links and joints connecting them in the form of URDF (non-articulated objects are composed of one link)
StableOrientations		A list of stable orientations assuming the object is placed on a flat surface, computed using a 3D geometry library
HeatSourceLink	heatSource	Virtual (non-colliding) fixed link that generates heat
FireSourceLink	fireSource	Virtual (non-colliding) fixed link that generates fire
CleaningToolLink	cleaningTool	Fixed link that needs to contact dirt particles for the tool to clean them
WaterSourceLink	waterSource	Virtual (non-colliding) fixed link that generates water
WaterSinkLink	waterSource	Virtual (non-colliding) fixed link that absorbs water
TogglingLink	toggleable	Virtual (non-colliding) fixed link that changes the toggled state of the object when contacted
SlicingLink	slicingTool	Fixed link that changes the sliced state of another object if it contacts it with enough force
RelevantJoints	openable	List of joints that are relevant to indicate whether an object is open
AttachmentKeypoints	assembleable	List of keypoint locations that will be connected with those of other objects when they are close enough
ClothKeypoints	foldable, unfoldable	List of keypoint locations are used to determine if the object is folded (if they are close enough) or unfolded (if they are far enough)
ContainerVolume	fillable	Virtual (non-colliding) fixed link that represents the inner volume of a fillable container.
Type	cloth, deformable, liquid, rope, physicalSubstance, visualSubstance	Type of the object, e.g. rigid body, fluid, physical/visual substance, cloth, deformable, which determines how it will be simulated in OMNIGIBSON

Table A.6: Permanent object model properties. Adapted from [59].

of the table. Other examples include setting the Temperature of an object to make it Frozen or the joint configuration of an object to make it Open. The details of the sampling functions for all the logical predicates can be found in Table A.8.

E.2 AMT Visual Realism Study

We conducted an Amazon Mechanical Turk (AMT) study to evaluate OMNIGIBSON's relative visual realism compared to multiple other simulation environments. We selected 50 representative 1280×720 images from OMNIGIBSON, AI2-Thor, ThreeDWorld, Habitat 2.0, and iGibson 2.0, and shuffled them randomly into 50 groups of 5 images, where each group contained a unique image from each simulation environment. For each group, participants were asked to rank the images in terms of visual realism, assigning the most realistic images of the group 1, and subsequent images $2, ..., 5$. Image shuffling was randomized between participants. When presenting our results, we invert the aggregated mean and standard deviation across 60 participants, such that a score of a 5 would represent the most visually realistic images.

Our criteria for selecting the images are the following: (a) we only included photos taken from *fully interactive* scenes for a fair comparison, and (b) rendering must come from within the simulation environment, without customized tuning (i.e. new users can expect this visual quality off-the-shelf with minimal adjustment).

E.3 Visual Modalities

OMNIGIBSON provides diverse perceptual modules to capture realistic sensor modalities from an agent's perspective, including RGB, Depth, Semantic Segmentation, Normal, and Optical Flow images, in addition to non-visual modalities such as proprioception and LiDAR scans (Fig. A.8). OMNIGIBSON also provides varying abstraction levels for agent action spaces, including low-level control, assistive manipulation, and primitive skill execution. Overall, these modules are intended to be useful to the broad embodied AI research community, with the goal of accelerating breakthroughs on BEHAVIOR-1K.

E.4 Performance Benchmark

To evaluate the performance of OMNIGIBSON under different conditions, we performed rigorous speed test in two representative scenes (Rs_int and restaurant_hotel) with different number of

Object State	Description and Update Rules
Pose	6 DoF pose (position and orientation) of the object in world reference frame, updated by the underlying physics engine.
AABB	Axis-aligned bounding box (coordinates of two opposite corners) of the object in the world reference frame, updated by the underlying physics engine.
JointStates	State of all internal DoFs of the (articulated) object for the structure defined by KinematicStructure, updated by the underlying physics engine.
ParticlePositions	Positions of all the underlying particles for cloth and fluid, updated by the underlying physics engine.
InContactObjs	List of all objects in physical contact with the object, updated by the underlying physics engine.
ConnectedObjs	List of all objects that are connected to the object, either via a fixed joint for rigid bodies or via an attachment for cloth and deformables.
InSamePositiveVerticalAxisObjs	List of all objects in the positive vertical axis drawn from the object's center of mass, updated by shooting a ray upwards in the positive z-axis and gather the objects hit by the ray.
InSameNegativeVerticalAxisObjs	List of all objects in the negative vertical axis drawn from the object's center of mass, updated by shooting a ray downwards in the negative z-axis and gather the objects hit by the ray.
InSameHorizontalPlaneObjs	List of all objects in the horizontal plane drawn from the object's center of mass, updated by shooting a number of ray in the x-y plane and gather the objects hit by the rays.
Temperature, T	Object's current temperature in $^\circ$C, updated by detecting if the object is affected by any heat source or heat sink.
MaxTemperature, T_{max}	Maximum temperature of the object reached historically during this simulation run, updated by keeping track of all the Temperature in the history.
SoakedLevel, w	Amount of liquid absorbed by the object corresponding to the number of liquid particles contacted, updated by detecting if the object is in contact with any liquid particle. This is maintained for every type of liquid separately.
CoveredLevel, c	Amount of visualSubstance that covers the object, updated by detecting if the particles of the visualSubstance are in contact with anything that can potentially remove them from the object, e.g. cleaningTool. This is maintained for every type of visualSubstance separately.
ToggledState, *TS*	Binary state indicating if the object is currently on or off, updated by detecting if the agent is in contact with the TogglingLink.
SlicedState, *SS*	Binary state indicating whether the object has been sliced (irreversible), updated by detecting if the object is in contact with any SlicingTool that exerts a force above a certain threshold F_{sliced}. We assume as default force threshold of $F_{sliced} = 10$ N, a value that can be configured per object category and model.
BrokenState, *BS*	Binary state indicating if the object is broken, updated by detecting if the object has a contact force with any other object above a certain threshold F_{broken}, a value that can be configured per object category and model.

Table A.7: Object states maintained by OMNIGIBSON. Adapted from [59].

objects on a single-GPU, single-process setup. We adopt the "idle" setup from Li et al. [59] and Szot et al. [26]: a robot is placed in the scene (except the last row "- Robot"), and stays still with zero velocity action. At each time step, the simulator runs the physics simulation, extended object state update, and transition machine update loop, and renders a 128×128 RGB image. We use action time step of $t_a = \frac{1}{30}$s and physics time step of $t_s = \frac{1}{120}$s to be consistent with previous works. Our benchmark runs on a Ubuntu machine with Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz and one Nvidia GeForce 2080 Ti GPU in a single process setting. The results are summarized in Table A.10.

OMNIGIBSON runs at a comparable speed to previous works like iGibson 2.0 [59] under similar settings, but with much higher rendering quality thanks to ray-tracing. It maintains a reasonable speed even for a large scene with over 800 objects. OMNIGIBSON is also highly configurable and provides flexible interface for users to balance between simulation fidelity and speed given their use cases and research interests. If the user isn't interested in fluid or cloth in their tasks, they can turn off these features to harvest performance speedup. Similarly, if the user is only interested in kinematics-only rearrangement tasks, or non-robotics embodied AI applications (e.g. a virtual camera), they can turn off object state update, or remove the robot, respectively. We haven't conducted any performance optimization, and we are actively working on improving aspects that can provide immediate and significant speedups, e.g. in areas like object sleeping, mesh simplification, more efficient object state update. Furthermore, since real-time ray-tracing and fluid/cloth simulation is still an active area of research, we expect the upcoming hardware and software advances from Nvidia will lead to significant performance improvements in these areas.

Predicate	Description
InsideOf(o_1,o_2)	Object o_1 is inside of object o_2 if we can find two orthogonal axes crossing at o_1 center of mass that intersect o_2 collision mesh in both directions.
OnTopOf(o_1,o_2)	Object o_1 is on top of object o_2 if $o_2 \in$ InSameNegativeVerticalAxisObjs$(o_1) \wedge o_2 \notin$ InSamePositiveVerticalAxisObjs$(o_1) \wedge$ InContactWith(o_1, o_2), where InSamePositive/NegativeVerticalAxisObjs(o_1) is the list of objects in the same positive/negative vertical axis as o_1 and InContactWith(o_1, o_2) is whether the two objects are in physical contact.
NextTo(o_1,o_2)	Object o_1 is next to object o_2 if $o_2 \in$ InSameHorizontalPlaneObjs$(o_1) \wedge l2(o_1, o_2) < t_{NextTo}$, where InSameHorizontalPlaneObjs(o_1) is a list of objects in the same horizontal plane as o_1, $l2$ is the L2 distance between the closest points of the two objects, and t_{NextTo} is a distance threshold that is proportional to the average size of the two objects.
InContactWith(o_1,o_2)	Object o_1 is in contact with o_2 if their surfaces are in contact in at least one point, i.e., $o_2 \in$ InContactObjs(o_1).
ConnectedWith(o_1,o_2)	Object o_1 is connected with o_2 if $o_2 \in$ ConnectedObjs(o_1).
Under(o_1,o_2)	Object o_1 is under object o_2 if $o_2 \in$ InSamePositiveVerticalAxisObjs$(o_1) \wedge o_2 \notin$ InSameNegativeVerticalAxisObjs(o_1).
OnFloor(o_1,o_2)	Object o_1 is on the room floor o_2 if InContactWith(o_1, o_2) and o_2 is of Room type.
Open(o)	Any joints (internal articulated degrees of freedom) of object o are open. Only joints that are relevant to consider an object Open are used in the predicate computation, e.g. the door of a microwave but not the buttons and controls. To select the relevant joints, object models of categories that can be Open undergo an additional annotation that produces a RelevantJoints list. A joint is considered open if its joint state q is 5% over the lower limit, i.e. $q > 0.05(q_{UpperLimit} - q_{LowerLimit}) + q_{LowerLimit}$.
Cooked(o)	The temperature of object o was over the cooked threshold, T_{cooked}, and under the burnt threshold, T_{burnt}, at least once in the history of the simulation episode, i.e., $T_{cooked} \leq T_o^{max} < T_{burnt}$. We annotate the cooked temperature T_{cooked} for each object category that can be Cooked.
Burnt(o)	The temperature of object o was over the burnt threshold, T_{burnt}, at least once in the history of the simulation episode, i.e., $T_o^{max} \geq T_{burnt}$. We annotate the burnt temperature T_{burnt} for each object category that can be Burnt.
OnFire(o)	The temperature of object o is above the on-fire threshold, T_{onfire}, i.e., $T_o \leq T_{onfire}$. We assume as default on-fire temperature $T_{onfire} = 300^\circ C$, a value that can be adapted per object category and model.
Frozen(o)	The temperature of object o is under the freezing threshold, T_{frozen}, i.e., $T_o \leq T_{frozen}$. We assume as default freezing temperature $T_{frozen} = 0^\circ C$, a value that can be adapted per object category and model.
Heated(o)	The temperature of object o is above the heated threshold, T_{heated}, i.e., $T_o \leq T_{heated}$. We assume as default heated temperature $T_{heated} = 75^\circ C$, a value that can be adapted per object category and model.
Boiled(l)	The temperature of liquid l is above the boiling point, T_{boiled}, i.e., $T_o \leq T_{boiled}$. We assume as default boiling point $T_{boiling} = 100^\circ C$, a value that can be adapted per object category and model.
Soaked(o, l)	The soaked level w of the liquid l for the object o is over a threshold, w_{soaked}, i.e., $w \geq w_{soaked}$. The default value for the threshold is $w_{soaked} = 50$, (the object is soaked if it absorbs more than 50 liquid particles), a value that can be adapted per object category and model and per liquid type.
Filled(o, l)	Object o is filled by liquid l if the number of particles of l that is inside the ContainerVolume of o is above a certain threshold percentage of the total volume. The default value for the threshold is $w_{filled} = 0.5$.
Covered(o, s)	For visualSubstance s, check if the covered level c of s for the object o is over a threshold, $c_{covered}$, i.e., $c \geq c_{covered}$; for physicalSubstance s, check if the number of particles of s that are in contact with the object o is over the same threshold. The default value for the threshold is $c_{covered} = 50$ (50 substance particles), a value that can be adapted per object category and model, and per substance.
ToggledOn(o)	Object o is toggled on or off. It is a direct query of the object's extended state TS, the toggled state.
Sliced(o)	Object o is sliced or not. It is a direct access of the object's extended state SS, the sliced state.
Broken(o)	Object o is broken or not. It is a direct access of the object's extended state BS, the broken state.
Folded(o)	Object o is folded if its corresponding ClothKeypoints are sufficiently close to each other.
Unfolded(o)	Object o is unfolded if its corresponding ClothKeypoints are sufficiently far from each other.
Assembled(o)	Object o is assembled if all of its parts have been correctly connected: every pair of parts p_i and p_j are connected (or not) with each other (i.e. IsConnected(p_i, p_j)) in a specific way defined by each object model.
Hung(o_1,o_2)	Object o_1 is hung onto object o_2 if they are connected, IsConnected(o_1, o_2).
Blended(o_1 ... o_n)	Objects o_1 to o_n are blended if they are in contact with each other, i.e. InContactWith(o_i, o_j) for all pairs.
InFoVOfAgent(o)	Object o is in the field of view of the agent, i.e., at least one pixel of the image acquired by the agent's onboard sensors corresponds to the surface of o.
InHandOfAgent(o)	Object o is grasped by the agent's hands (i.e. assistive grasping is activated on that object).
InReachOfAgent(o)	Object o is within $d_{reach} = 2$ meters away from the agent.
InSameRoomAsAgent(o)	Object o is located in the same room as the agent.

Table A.8: **Logical Predicates:** Description of the checking functions. Adapted from [59].

Predicate	Sampling Mechanism
InsideOf(o_1,o_2)	Only InsideOf(o_1,o_2) = True can be sampled. o_1 is randomly sampled within o_2 using a ray-casting mechanism adopted from [27]. o_1 is guaranteed to be supported fully by a surface and free of collisions with any other object except o_2.
OnTopOf(o_1,o_2)	Only OnTopOf(o_1,o_2) = True can be sampled. o_1 is randomly sampled on top of o_2 using a ray-casting mechanism adopted from [27]. o_1 is guaranteed to be supported fully by a surface and free of collisions with any other object except o_2.
ConnectedWith(o_1,o_2)	Create a rigid joint between the two objects if they are both rigid bodies, or an attachment between them otherwise.
Under(o_1,o_2)	Only Under(o_1,o_2) = True can be sampled. o_1 is randomly sampled on top of the floor region beneath o_2 using a ray-casting mechanism adopted from [27]. o_1 is guaranteed to be supported fully by a surface and free of collisions with any other object except the floor.
OnFloor(o_1,o_2)	Only OnFloor(o_1,o_2) = True can be sampled. o_1 is randomly sampled on top of o_2, which is the floor of a certain room, using the scene's room segmentation mask. o_1 is guaranteed to be supported fully by a surface and free of collisions with any other object except o_2.
Open(o)	To sample an object o with the predicate Open(o) = True, a subset of the object's relevant joints (using the RelevantJoints model property) are selected, and each selected joint is moved to a uniformly random position between the openness threshold and the joint's upper limit. To sample an object o with the predicate Open(o) = False, all of the object's relevant joints (using the RelevantJoints model property) are moved to a uniformly random position between the joint's lower limit and the openness threshold.
Cooked(o)	To sample an object o with the predicate Cooked(o) = True, the object's MaxTemperature is updated to $\max(T_o^{max}, T_{cooked})$. Similarly, to sample an object o with the predicate Cooked(o) = False, the object's MaxTemperature is updated to $\min(T_o^{max}, T_{cooked} - 1)$.
Burnt(o)	To sample an object o with the predicate Burnt(o) = True, the object's MaxTemperature is updated to $\max(T_o^{max}, T_{burnt})$. Similarly, to sample an object o with the predicate Cooked(o) = False, the object's MaxTemperature is updated to $\min(T_o^{max}, T_{burnt} - 1)$.
OnFire(o)	To sample an object o with the predicate OnFire(o) = True, the object's Temperature is updated to a uniformly random temperature between $T_{onfire} + 10$ and $T_{onfire} + 50$. To sample an object o with the predicate OnFire(o) = False, the object's Temperature is updated to $T_{onfire} - 1$.
Frozen(o)	To sample an object o with the predicate Frozen(o) = True, the object's Temperature is updated to a uniformly random temperature between $T_{frozen} - 10$ and $T_{frozen} - 50$. To sample an object o with the predicate Frozen(o) = False, the object's Temperature is updated to $T_{frozen} + 1$.
Heated(o)	To sample an object o with the predicate Heated(o) = True, the object's Temperature is updated to a uniformly random temperature between $T_{heated} + 10$ and $T_{heated} + 50$. To sample an object o with the predicate Heated(o) = False, the object's Temperature is updated to $T_{heated} - 1$.
Boiled(l)	To sample a liquid type l with the predicate Boiled(l) = True, the Temperature of all particles of l is updated to a uniformly random temperature between $T_{boiled} + 10$ and $T_{boiled} + 50$. To sample a liquid type l with the predicate Boiled(o) = False, the Temperature of all particles of l is updated to $T_{boiled} - 1$.
Soaked(o, l)	To sample an object o and a liquid type l with the predicate Soaked(o, l) = True, the object's SoakedLevel w for l is updated to match the Soaked threshold of w_{soaked}. To sample an object o and a liquid type l with the predicate Soaked(o, l) = False, the object's SoakedLevel w is updated to 0.
Filled(o, l)	To sample an object o and a liquid type l with the predicate Filled(o, l) = True, an appropriate number of particles of l are sampled inside the ContainerVolume of o so that they fill up enough of its volume ($\geq w_{filled} = 0.5$). To sample an object o and a liquid type l with the predicate Filled(o, l) = False, all particles of l that are inside the ContainerVolume of o (if any) are removed.
Covered(o, s)	To sample an object o and a substance s with Covered(o, s) = True, a fixed number of particles of s are randomly placed on the surface of o using a ray-casting mechanism adopted from [27]. To sample an object o and a visualSubstance s with Covered(o, s) = False, all particles of s is removed from o and the corresponding CoveredLevel is set to 0.
ToggledOn(o)	The ToggledState of the object is updated to match the required predicate value.
Sliced(o)	The SlicedState of the object is updated to match the required predicate value. Also, the whole object are replaced with the two halves, that will be placed at the same location and inherit the extended states from the whole object (e.g. Temperature).
Broken(o)	The BrokenState of the object is updated to match the required predicate value. Also, the whole object is broken down into pieces, that will be placed at the same location.

Table A.9: **Logical Predicates:** Description of the sampling functions. Adapted from [59].

F Baselines Details

In this section, we include additional information about the baselines evaluated and analyzed in the main paper: the visuomotor control baseline and two variants of the baselines using action primitives, with and without the history of observations.

F.1 Network Architecture

For RL-Prim. and RL-Prim.Hist. that use PPO as the underlying RL algorithm, the architecture consists of a visual feature extractor that takes the egocentric visual observation as input and a state encoder neural network which takes whether the robot is grasping an object or not as input. The image

Eval. Conditions	Scene	
	Rs_int (60 objs)	restaurant_hotel (808 objs)
Full Feature Set	13	5
- Fluid and Cloth	63	16
- Object State Update	119	47
- Robot	232	54

Table A.10: Benchmarking OMNIGIBSON performance: simulation steps per second (SPS, higher better) in two representative scenes with 60 and 808 objects, under different evaluation conditions.

input is normalized using a per-channel moving average. These two encodings are passed into an MLP module, which is then processed by a value head to predict the value for the given observations and an action head to produce a discrete action that corresponds to an action primitive executed on an object. The size of input images is $128 \times 128 \times 3$. The feature extractor is a sequential architecture of Conv-ReLU-MaxPooling-Flatten. MLP converts the feature into a 128-dimensional vector. The details of RL-Prim.'s network architecture is illustrated in Fig. A.11 (b). For RL-VMC that uses SAC as the underlying RL algorithm, we use the same feature extractor as RL-Prim.. RL-VMC consists of an actor network, a critic network, and a target critic network. All of them are MLPs with ReLU activation functions. Fig. A.11 (a) illustrates the details of RL-VMC's network architecture. RL-VMC has the continuous action space and outputs low-level control actions directly.

F.2 Task settings

Fig. A.9 depicts the three activities we consider in our experiments:

- `StoreDecoration`: a tidying activity where the agent must pick up and store Halloween decorations into a cabinet operating articulated objects. Action space: `navigate`, `pick`, `place`, `push`. The goal is to store two pumpkins into a drawer (see Fig. A.9 (a)).
- `CollectTrash`: A collecting activity that requires the agent to gather empty bottles and cups and throw them into a trash bin. Action space: `navigate`, `pick`, `place`. The goal is to throw two bottles and two cups into a trash bin (see Fig. A.9 (b)).
- `CleanTable`: A cleaning activity that involves challenging cloth manipulation and fluids for table cleaning. Action space: `navigate`, `pick`, `dip`, `wipe`. The goal is to cleaning a table with a soaked cloth (see Fig. A.9 (c)).

Each activity takes place in a different B1K scene, `Rs_int`, `mockup_apt`, and `restaurant_hotel`.

F.3 Training details

Learning objectives. For RL-Prim. and RL-Prim.Hist., we use an on-policy reinforcement learning algorithm Proximal Policy Optimization (PPO). θ is the policy parameter, $\hat{E}_t$ denotes the empirical expectation over timesteps. r_t is the ratio of the probability under the new and old policies, respectively. $\hat{A}_t$ is the estimated advantage at time t. ϵ is the clipping hyperparameter. The objective function of PPO is shown in Equation 1:

$$C^{CLIP}(\theta) = \hat{E}_t[\min r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t]. \tag{1}$$

For RL-VMC, we use Soft Actor Critic (SAC) algorithm. The objective function is shown in Equation 2:

$$\pi^* = \arg\max_{\pi} E_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\left(R(s_t, a_t, s_{t+1}) + \alpha H(\pi(\cdot|s_t))\right)\right], \tag{2}$$

where α is the coefficient that determines the weight of two terms and H denotes the entropy.

Reward function. We use the success signal provided by the BDDL activity definition as the reward function for training the policy in all methods. For instance, in the `StoreDecorations` task, the BDDL task goal definition is `Forall(decoration){Inside(decoration, cabinet)}`. The agent receive a positive signal if and only if one of the `decoration` is physically placed inside the `drawer` of the `cabinet`. This is a challenging sparse reward setting since the agent needs to plan multiple steps to reach the final goal without any intermediate subgoal rewards (e.g. `push` open the `drawer` must take place before `pick` and `place` the `decoration`, but no reward signal will be given to the `push` open action).

Learning Rate	0.0003
Buffer Size	300
Batch Size	64
Discount (γ)	0.99
Soft Update Coefficient	0.005

Table A.11: Hyperparameters of SAC for the RL-VMC baseline

Learning Rate	0.0003
Buffer Size	300
Batch Size	64
Discount (γ)	0.99
GAE Parameter γ_{gae}	0.99
Clipping Parameter ϵ	0.2
Entropy Coeff c_1	0.0
VF Coeff c_2	0.5

Table A.12: Hyperparameters of PPO for the RL-Prim. and RL-Prim.Hist. baselines

Hyperparameters. The hyperparameters for SAC and PPO in the baselines are summarized in Table A.11 and Table A.12. Both SAC and PPO are trained for 30,000 time steps. We train with seeds $0, 1, 2$ and evaluate with seed 0.

Computation. For each run of our experiments, we use either a single Nvidia GeForce RTX 2080 Ti or a single Nvidia RTX A-6000 GPU, together with Intel Xeon CPU, and 40GB of RAM. During training, the GPU memory usage is around 9GB. Depending on which task, the total training iterations range between 10k to 25k, and the total wall-clock training time range between 3.75 to 7.5 hours.

F.4 Action Primitives

BEHAVIOR-1K activities are long-horizon which require hundreds if not thousands of environment steps (low-level robot control signals) to be completed. A way to overcome this challenge (e.g., reinforcement learning) is to modify the original action space with a set of *action primitives*, i.e., time-extended actions that correspond to multiple low-level commands and that achieve some expected outcome, such as grasping an object or navigating to a location. For our baselines, we defined six of these primitives, and implemented them using a sampling-based motion planner.

All the primitives are compositional: collision-free paths of the entire robot to navigate to a location, collision-free paths of the robot's arm to reach a `6D_pose` with the end-effector, or a predefined arm joint configuration, trajectories where the end-effector follows a `line` in Cartesian space, possibly colliding/interacting with the environment, and sequences of actions to `open`/`close` the robot's gripper while holding the position. All manipulation primitives start with a trajectory to move the arm from a tucked configuration (folded arm) to an untucked configuration to enable subsequent arm interaction, and end with the inverse motion, from untucked to tucked configuration, to allow subsequent collision-free navigation. The action primitives take as parameter an object to apply the action on; we assume access to the 3D position of the objects to obtain the necessary parameters to query the motion planner. This procedure is replicated on the real robot, where we combine detections from YOLO [70] with information from the depth map of the RGB-D images to obtain the parameters (see Sec. G). For the navigation actions, we assume a set of known relevant locations to navigate to (next to the rectangles in Fig. A.9)

The primitives we used in our experiments can be seen in Fig. A.10 and include:

a) `navigate`: Collision-free trajectory of the entire robot to a location.

b) `pick`: Composition of 1) a trajectory to a pre-grasp `6D_pose` above the object to pick, 2) a `line` trajectory in Cartesian space down towards the object, interrupted when there is contact, 3) a `closing` action, 4) a first retracting trajectory following a `line` upwards, and 5) a second retracting trajectory to reach the untucked joint configuration. The primitive only executes if the robot is not currently picking another object.

c) `place`: Composition of 1) a trajectory to a pre-place `6D_pose` above the object to place the grasped object on, 2) an `opening` action, 3) a second retracting trajectory to reach the untucked joint configuration. The primitive only executes if the robot is currently holding an object.

d) `push`: Composition of 1) a trajectory to a pre-push `6D_pose` above the object to push-open, 2) a `line` trajectory in Cartesian space down towards the object, interrupted if there is contact, 3) a `line` trajectory in Cartesian space to push-open the object, e.g., towards the robot, 4) a first retracting trajectory following a `line` upwards, and 5) a second retracting trajectory to reach the untucked joint configuration.

e) `dip`: Composition of 1) a trajectory to a pre-dip `6D_pose` above the object to dip into, 2) a `line` trajectory in Cartesian space down towards the object to dip, 3) a `line` trajectory in Cartesian

space upwards, and 4) a retracting trajectory to reach the initial joint configuration. The primitive only executes if the robot is currently holding an object.

f) `wipe`: Composition of 1) a trajectory to a pre-wipe `6D_pose` above the object to wipe, 2) a `line` trajectory in Cartesian space down towards the object to be wiped, interrupted when there is contact, 3) a `line` trajectory in Cartesian space horizontally to wipe left-right, 4) a `line` trajectory in Cartesian space horizontally to wipe towards the robot, 4) a first retracting trajectory following a `line` upwards, and 5) a second retracting trajectory to reach the untucked joint configuration. The primitive only executes if the robot is currently holding an object.

F.5 Additional Metrics: success score Q

The success score Q [27] for each task is reported in Table A.13. We observe the same trend in Q as in the success rate. This is another indication of the effectiveness of our approach.

Method	Policy Features		Success score Q		
	Primitives	History	`StoreDecoration`	`CollectTrash`	`CleanTable`
RL-VMC	✗	✗	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
RL-Prim.	✓	✗	0.50 ± 0.05	0.49 ± 0.04	0.77 ± 0.08
RL-Prim.Hist.	✓	✓	0.59 ± 0.03	0.68 ± 0.02	0.88 ± 0.02

Table A.13: Performance of three baseline methods, in terms of the success score Q.

G Details of the Real-World Setup

Scene. Our real-world experiments take place in a mockup studio-apartment in our lab that includes a bedroom, living room, and dining area. This real-world scene was modeled with a digital counterpart in simulation for BEHAVIOR-1K: the `mockup_apt` scene. Fig. 5 depicts both real and digital twin side-by-side. The digital model has been created by first, scanning the real-world apartment using a phone and commercial software [A11], then, replacing and projecting the texture of walls, floors, and ceilings, and finally, replacing the existing objects with 3D models from our BEHAVIOR-1K DATASET. This process should minimize the effect caused by differences between the real world and the 3D model. The simulated and the real robot use the same 2D map of the scene for localization. In this way, 2D locations in the real world correspond to the same 2D locations in simulation.

Robot platform. We use in our experiments a Tiago++ model from PAL Robotics, with an omnidirectional base, two 7-degrees-of-freedom arms with parallel-yaw grippers, a 1-degree-of-freedom prismatic torso, two SICK LiDAR sensors (back and front of the base), and an ASUS Xtion RGB-D camera mounted on the robot's head, which can be controlled in yaw and pitch. All sensors and actuators are connected through the Robot Operating System, ROS [A12]. The code runs on a laptop with an Nvidia GTX 1070 that sends the commands to the onboard robot computer to be executed.

Action primitives on the real robot. We implemented a version of the action primitives `navigate`, `pick` and `place` on the real robot similar to their implementation in OMNIGIBSON. For navigation, we use a 2D sampling-based motion planner on the 2D map we use for localization and that has been created from the 3D model of the apartment. For manipulation in the real world, we need to obtain the parameters necessary to plan the manipulation primitives (`pick` and `place`) from the sensor signals. To do that, we first obtain detections from a YOLO v3 [70] object detector on the RGB images from the robot's camera. If the robot attempts to execute an action primitive on an object that has not been detected, we return failure (*Object Detection Failure* in our analysis). If the object to manipulate has been detected, we query the pixels in the depth map corresponding to the 21×21 window at the center of the detection bounding box, and use this information to compute the centroid of the corresponding 3D points. This location will be used as an object location for grasping or placing. This information is enough to create a sequence of paths using a sampling-based motion planner (RRT [62]) similar to the ones used in simulation. The motion planner uses a voxel representation of the scene obtained from the depth sensor.

Experimental setup. We randomized three parameters in our real robot experiments: the initial location of the robot (chosen from the three possible activity locations), the positions of the objects on the table, and the orientations of the objects (upright or lying down). The experiment runs cover uniformly these parameters. Additionally, for vision-based policies, we evaluated two lighting

conditions: full lighting (including the ceiling) and only lamps. Failures are defined as follows: Motion planning failures occurred when the robot was unable to plan an end effector trajectory to complete the action primitive. This includes failures caused by navigation noise, where after navigating to the task location, the robot was too far away from the target object to execute the pick or place action. Grasping failures include errors that occurred during the execution of the grasp such as pushing the object off the table while attempting a grasp or losing the object because it slips out, as well as the robot dropping the object after grasping it. Placing failures include the robot dropping the object in the wrong location. Object detection failures primarily resulted from our object detector, YOLO v3 [70], failing to detect the object to interact with; a second, less common, object detection failure corresponds to the depth camera returning invalid measurements for the detected object. Finally, we consider policy failures when the policy selects the same invalid action three times in a row, e.g., requesting to navigate to the bin when it is already in front of the bin. The main policy error corresponds to the policy repeatedly choosing the same action primitive. We observed empirically that, even though the policy selection presents some small randomness when the policy requests the same invalid action three times in a row, most subsequent calls will be also the same invalid action and we decide to terminate early.

G.1 Further Characterization of the Sim-Real Gap

We performed additional experiments to characterize the gap between simulation and the real world. In our experiments, we moved the robot to different locations in the `mockup_apt` scene and collected real sensor signals: RGB images, depth maps, and LiDAR measurements. We then moved the robot in simulation to the same locations and collected virtual sensor signals. Some of the images are depicted in Fig. A.12. We observe that, thanks to our highly realistic models, OMNIGIBSON provides high-fidelity sensor signals that approximate the real-world ones. However, some of the sources of noise in the real sensors are not modeled currently in OMNIGIBSON, contributing to a sim-real gap, e.g., the poor dynamic range of the RGB camera, or the "shadow" effects in depth maps due to the projected light mechanism. This analysis indicates possible avenues to further close the sensor gap between simulation and the real world, e.g., by including sensor noise models in OMNIGIBSON or by leveraging sim-to-real techniques (e.g., domain randomization, system identification).

G.2 Additional Experiments

In addition to RL-Prim.Hist., we also evaluated RL-VMC and RL-Prim.Hist. in the real world and observed a similar trend of performance in the real world as in simulation: RL-VMC < RL-Prim. < RL-Prim.Hist.. RL-VMC still achieves zero success because of sparse reward and exploration difficulty during training. RL-Prim. has worse performance than RL-Prim.Hist.: on average, the robot with RL-Prim.Hist. successfully places 0.64 cups/bottles, whereas the one with RL-Prim. places 0. Qualitatively, RL-Prim. tends to get stuck in a repetitive action loop because the agent is unaware of its action history. This preliminary result offers us some confidence that OMNIGIBSON can be a reliable test bed for future sim-to-real robotics research.

H Ethical Statement

BEHAVIOR-1K aims to drive embodied AI solutions that fulfill human need. A primary ethical consideration of BEHAVIOR-1K is therefore the method used to determine such need. To understand which activities would be useful for humans, we survey 1,461 respondents on Amazon Mechanical Turk and collected 50 responses for each activity, aiming for a clear consensus signal. Though this is a large group, the results are still biased by the fact that the researchers, annotators, and data providers only represent a small population relative to potential users of such technology. We plan to address these biases by making BEHAVIOR-1K DATASET open-source and invite a wider community to contribute to its knowledge base.

Furthermore, compared to the U.S. population, the demographic representation in our survey skews white, male, non-disability status, mid-five figure incomes, and 30-40 age range with more representation on the higher side than the lower side of that range—closer to the demographics of Mechanical Turk workers. This also biases the survey population toward certain responses, creating potential ethical limitations.

Specifically this bias may affect the survey's representation of people who would be affected most by autonomous agents [A13]. We therefore asked several explicit questions such as "Do you do household work for a living?", "If you do household work for a living, would you benefit from

assistance?", "Do you pay for someone else to do household work for you?". Future work involves using these questions to direct benchmark development.

Appendix References

[A1] Mahnaz Koupaee and William Yang Wang. Wikihow: A large scale text summarization dataset. *arXiv preprint arXiv:1810.09305*, 2018.

[A2] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[A3] Douglas C. Montgomery. *Design and analysis of experiments*. John Wiley & Sons, Inc., Hoboken, NJ, eighth edition, 2013.

[A4] Gabriele Paolacci, Jesse Chandler, and Panagiotis G Ipeirotis. Running experiments on amazon mechanical turk. *Judgment and Decision making*, 5(5):411–419, 2010.

[A5] Pew Research Center. Research in the crowdsourcing age, a case study. Technical report, Washington, D.C., July 2016.

[A6] Alan Akbik, Duncan Blythe, and Roland Vollgraf. Contextual string embeddings for sequence labeling. In *COLING 2018, 27th International Conference on Computational Linguistics*, pages 1638–1649, 2018.

[A7] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, dec 2003.

[A8] Julia Wichlacz, Álvaro Torralba, and Jörg Hoffmann. Construction-planning models in minecraft. In *Proceedings of the 2nd ICAPS Workshop on Hierarchical Planning (HPlan 2019)*, pages 1–5, 2019.

[A9] Alex Oliver and Timothy Smiley. Multigrade predicates. *Mind*, 113(452):609–681.

[A10] TurboSquid, Inc. Turbosquid. https://www.turbosquid.com/, 2022. Accessed: 2022-06-24.

[A11] Niantic, Inc. Scaniverse. https://scaniverse.com, 2022. Accessed: 2022-06-24.

[A12] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[A13] David Baboolall, Duwain Pinder, and Shelley Stewart. How automation could affect employment for women in the united kingdom and minorities in the united states. *McKinsey Digital*.

Listing 1: MakeScones

```
(define
  (problem making_scones_1)
  (:domain omnigibson)

  (:objects
    flour.n.01_1 - flour.n.01
    sack.n.01_1 - sack.n.n01
    sugar.n.01_1 - sugar.n.01
    jar.n.01_1 - jar.n.01
    egg.n.02_1 egg.n.02_2 - egg.n.02
    milk.n.01_1 - milk.n.01
    bottle.n.01_1 - bottle.n.01
    butter.n.01_1 - butter.n.01
    bowl.n.01_1 - bowl.n.01
    baking_powder.n.01_1 - baking_powder.n.01
    box.n.01_1 box.n.01_2 - box.n.01
    scone.n.01_1 scone.n.01_2 scone.n.01_3
      scone.n.01_4 scone.n.01_5 scone.n.01_6
      - scone.n.01
    mixer.n.04_1 - mixer.n.04
    mixing_bowl.n.01_1 - mixing_bowl.n.01
    baking_tray.n.01_1 - baking_tray.n.01
    electric_refrigerator.n.01_1
      - electric_refrigerator.n.01
    cabinet.n.01_1 cabinet.n.01_2 - cabinet.n.01
    countertop.n.01_1 - countertop.n.01
    floor.n.01_1 - floor.n.01
    agent.n.01_1 - agent.n.01
  )

  (:init
    (filled flour.n.01_1 sack.n.01_1)
    (ontop sack.n.01_1 countertop.n.01_1)
    (filled sugar.n.01_1 jar.n.01_1)
    (inside jar.n.01_1 cabinet.n.01_1)
    (inside egg.n.02_1 box.n.01_1)
    (inside egg.n.02_2 box.n.01_1)
    (inside box.n.01_1
      electric_refrigerator.n.01_1)
    (filled milk.n.01_1 bottle.n.01_1)
    (inside bottle.n.01_1
      electric_refrigerator.n.01_1)
    (filled butter.n.01_1 bowl.n.01_1)
    (inside bowl.n.01_1
      electric_refrigerator.n.01_1)
    (filled baking_powder.n.01_1 box.n.01_2)
    (inside box.n.01_2 cabinet.n.01_1)
    (ontop mixer.n.04_1 countertop.n.01_1)
    (inside mixing_bowl.n.01_1 cabinet.n.01_2)
    (inside baking_tray.n.01_1 cabinet.n.01_2)
    (future scone.n.01_1)
    (future scone.n.01_2)
    (future scone.n.01_3)
    (future scone.n.01_4)
    (future scone.n.01_5)
    (future scone.n.01_6)
    (inroom oven.n.01_1 kitchen)
    (inroom countertop.n.01_1 kitchen)
    (inroom cabinet.n.01_1 kitchen)
    (inroom cabinet.n.01_2 kitchen)
    (inroom floor.n.01_1 kitchen)
    (onfloor floor.n.01_1 agent.n.01_1)
  )

  (:goal
    (and
      (cooked scone.n.01_1)
      (cooked scone.n.01_2)
      (cooked scone.n.01_3)
      (cooked scone.n.01_4)
      (cooked scone.n.01_5)
      (cooked scone.n.01_6)
    )
  )
)
```

Listing 2: CleanYourLaundryRoom

```
(define
  (problem clean_your_laundry_room_1)
  (:domain omnigibson)

  (:objects
    rag.n.01_1 - rag.n.01
    dryer.n.01_1 - dryer.n.01
    water.n.06_1 - water.n.06
    vinegar.n.01_1 - vinegar.n.01
    washer.n.03_1 - washer.n.03
    dust.n.01_1 - dust.n.01
    mold.n.05_1 - mold.n.05
    floor.n.01_1 - floor.n.01
    agent.n.01_1 - agent.n.01
  )

  (:init
    (ontop rag.n.01_1 dryer.n.01_1)
    (not
      (covered water.n.06_1 rag.n.01_1)
    )
    (empty vinegar.n.01_1 washer.n.03_1)
    (covered dust.n.01_1 dryer.n.01_1)
    (covered mold.n.05_1 washer.n.03_1)
    (onfloor agent.n.01_1 floor.n.01_1)
    (filled water.n.06_1 bottle.n.01_1)
    (onfloor bottle.n.01_1 floor.n.01_1)
    (inroom washer.n.03_1 laundry_room)
    (inroom floor.n.01_1 laundry_room)
  )

  (:goal
    (and
      (filled ?vinegar.n.01_1 ?washer.n.03_1)
      (not
        (covered ?dust.n.01_1 ?dryer.n.01_1)
      )
      (not
        (covered ?mold.n.05_1 ?washer.n.03_1)
      )
    )
  )
)
```

Listing 3: FixingMailbox

```
(define
  (problem fixing_mailbox_1)
  (:domain omnigibson)

  (:objects
    mailbox.n.01_1 - mailbox.n.01
    rust.n.01_1 - rust.n.01
    hammer.n.02_1 - hammer.n.02
    nail.n.02_1 nail.n.02_2 nail.n.02_3
      nail.n.02_4 nail.n.02_5 nail.n.02_6
      - nail.n.02
    emery_paper.n.01_1 - emery_paper.n.01
    lawn.n.01_1 - lawn.n.01
    floor.n.01_1 - floor.n.01
    agent.n.01_1 - agent.n.01
  )

  (:init
    (broken mailbox.n.01_1)
    (covered rust.n.01_1 mailbox.n.01_1)
    (ontop hammer.n.02_1 lawn.n.01_1)
    (ontop nail.n.02_1 lawn.n.01_1)
    (ontop nail.n.02_2 lawn.n.01_1)
    (ontop nail.n.02_3 lawn.n.01_1)
    (ontop nail.n.02_4 lawn.n.01_1)
    (ontop nail.n.02_5 lawn.n.01_1)
    (ontop nail.n.02_6 lawn.n.01_1)
    (ontop mailbox.n.01_1 lawn.n.01_1)
    (ontop emery_paper.n.01_1 lawn.n.01_1)
    (inroom lawn.n.01_1 garden)
    (inroom floor.n.01_1 garden)
    (onfloor agent.n.01_1 floor.n.01_1)
  )

  (:goal
    (and
      (not
        (covered ?rust.n.01_1 ?mailbox.n.01_1
      )
      (not
        (broken ?mailbox.n.01_1)
      )
    )
  )
)
```

Listing 4: PackingLunch

```
(define
  (problem packing_lunches_1)
  (:domain igibson)

  (:objects
    shelf.n.01_1 - shelf.n.01
    water.n.06_1 - water.n.06
    countertop.n.01_1 - countertop.n.01
    apple.n.01_1 - apple.n.01
    electric_refrigerator.n.01_1 -
        electric_refrigerator.n.01
    hamburger.n.01_1 - hamburger.n.01
    basket.n.01_1 - basket.n.01
  )

  (:init
    (ontop water.n.06_1 countertop.n.01_1)
    (inside apple.n.01_1
        electric_refrigerator.n.01_1)
    (inside hamburger.n.01_1
        electric_refrigerator.n.01_1)
    (ontop basket.n.01_1 countertop.n.01_1)
    (inroom countertop.n.01_1 kitchen)
    (inroom electric_refrigerator.n.01_1
        kitchen)
    (inroom shelf.n.01_1 kitchen)
  )

  (:goal
    (and
      (for_n_pairs
        (1)
        (?hamburger.n.01 - hamburger.n.01)
        (?basket.n.01 - basket.n.01)
        (inside ?hamburger.n.01 ?basket.n.01)
      )
      (for_n_pairs
        (1)
        (?basket.n.01 - basket.n.01)
        (?water.n.06 - water.n.06)
        (inside ?water.n.06 ?basket.n.01)
      )
      (for_n_pairs
        (1)
        (?basket.n.01 - basket.n.01)
        (?apple.n.01 - apple.n.01)
        (inside ?apple.n.01 ?basket.n.01)
      )
      (forall
        (?basket.n.01 - basket.n.01)
        (ontop ?basket.n.01 ?countertop.n.01_1)
      )
    )
  )
)
```

Listing 5: ServingHorsDoeuvres

```
(define
  (problem serving_hors_d_oeuvres_1)
  (:domain igibson)

  (:objects
    tray.n.01_1 tray.n.01_2 - tray.n.01
    countertop.n.01_1 - countertop.n.01
    oven.n.01_1 - oven.n.01
    sausage.n.01_1 sausage.n.01_2 - sausage.n.01
    cherry.n.03_1 cherry.n.03_2 - cherry.n.03
    electric_refrigerator.n.01_1 -
        electric_refrigerator.n.01
  )

  (:init
    (ontop tray.n.01_1 countertop.n.01_1)
    (ontop tray.n.01_2 countertop.n.01_1)
    (inside sausage.n.01_1 oven.n.01_1)
    (inside sausage.n.01_2 oven.n.01_1)
    (inside cherry.n.03_1
        electric_refrigerator.n.01_1)
    (inside cherry.n.03_2
        electric_refrigerator.n.01_1)
    (inroom oven.n.01_1 kitchen)
    (inroom electric_refrigerator.n.01_1
        kitchen)
    (inroom countertop.n.01_1 kitchen)
  )

  (:goal
    (and
      (exists
        (?tray.n.01 - tray.n.01)
        (and
          (forall
            (?sausage.n.01 - sausage.n.01)
            (ontop ?sausage.n.01 ?tray.n.01)
          )
          (forall
            (?cherry.n.03 - cherry.n.03)
            (not
              (ontop ?cherry.n.03 ?tray.n.01)
            )
          )
        )
      )
      (exists
        (?tray.n.01 - tray.n.01)
        (and
          (forall
            (?cherry.n.03 - cherry.n.03)
            (ontop ?cherry.n.03 ?tray.n.01)
          )
          (forall
            (?sausage.n.01 - sausage.n.01)
            (not
              (ontop ?sausage.n.01 ?tray.n.01)
            )
          )
        )
      )
    )
  )
)
```

Figure A.3: Part 1 of a scene collage showing views of all 50 scenes.

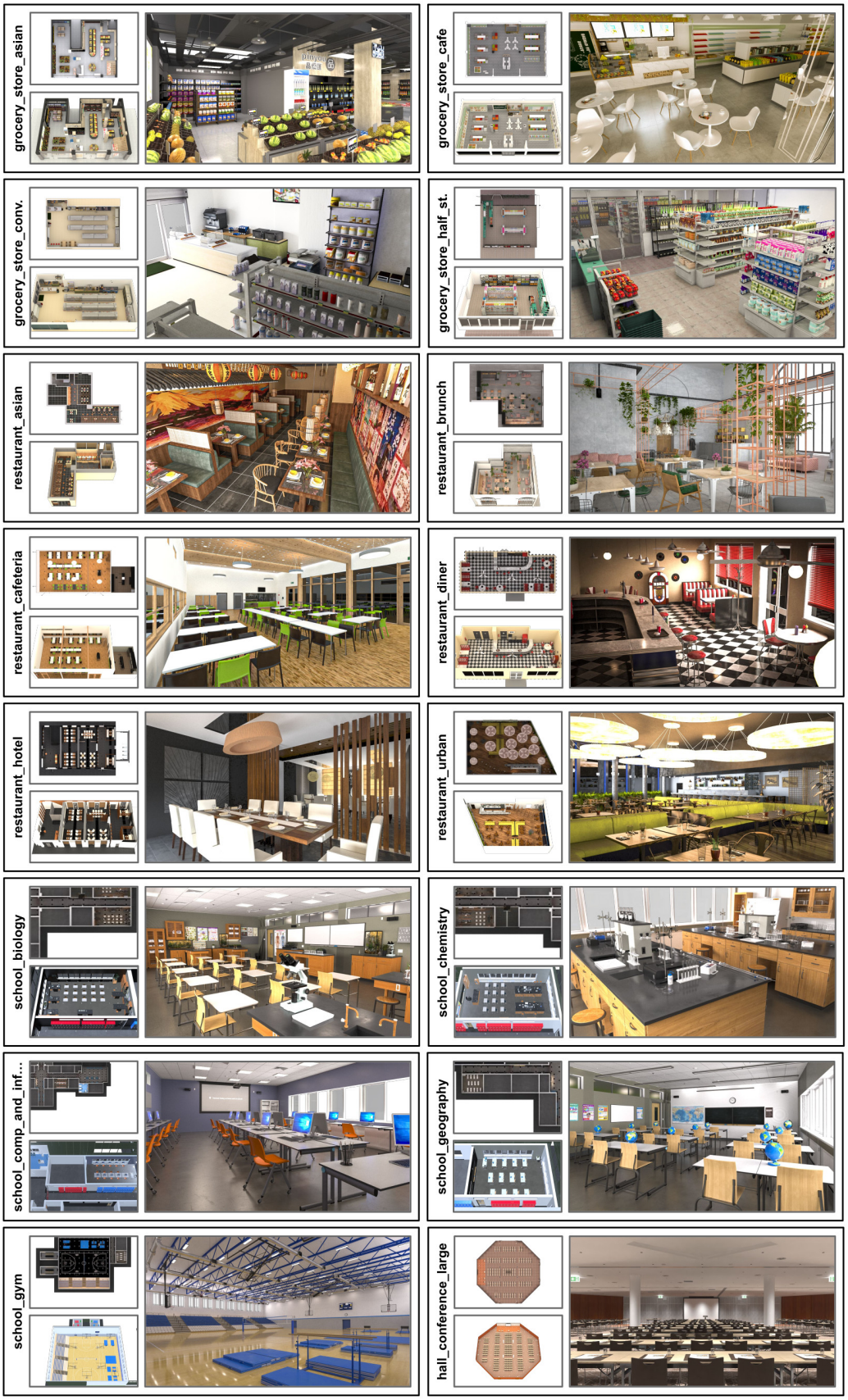


Figure A.4: Part 2 of a scene collage showing views of all 50 scenes.

Figure A.5: Part 3 of a scene collage showing views of all 50 scenes.

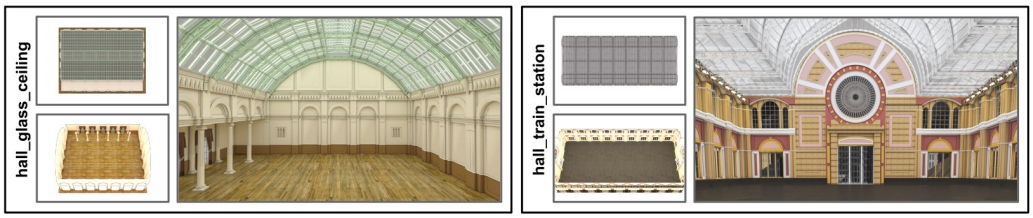


Figure A.6: Part 4 of a scene collage showing views of all 50 scenes.

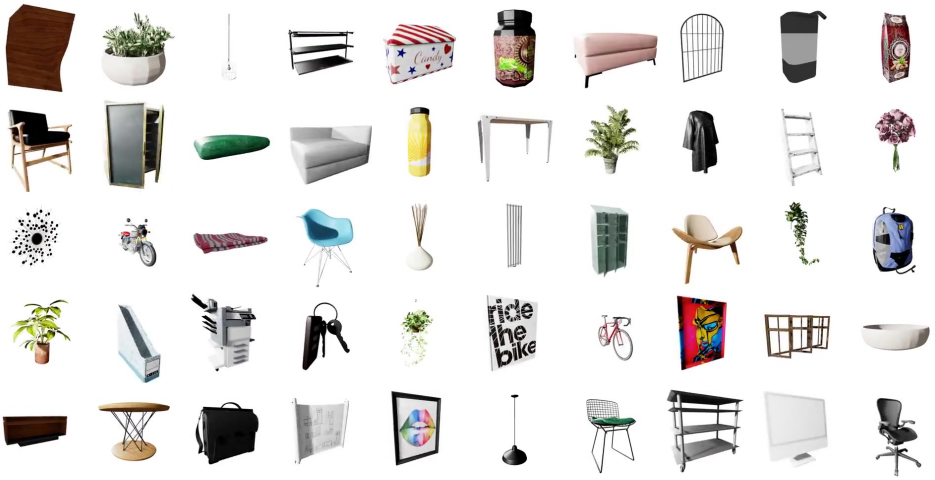


Figure A.7: A collage of objects included in the BEHAVIOR-1K DATASET. These objects highlight key features of OMNIGIBSON such as transparency, articulation, heat simulation, and lighting.

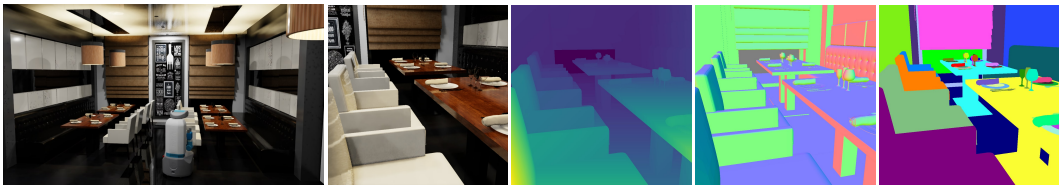


Figure A.8: **Visual modalities provided by OMNIGIBSON.** The robot observes the scene (left, 3rd person view) and obtains visual observations from its onboard sensors including RGB images, depth, normals, and instance segmentation. Rendered RGB images are highly realistic, and, combined with the diverse set of visual observations, can be used to train visual policies.

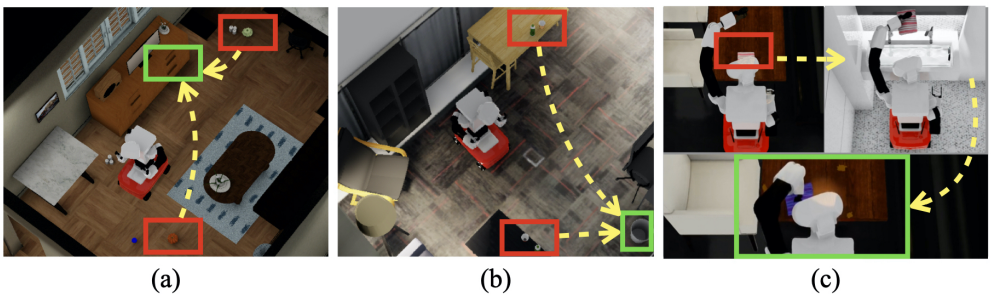


Figure A.9: **Activities used in our evaluation.** From left to right: (a). `StoreDecoration`, (b). `CollectTrash` and (c). `CleanTable`. The rectangles indicate relevant locations for the activities, e.g., locations with objects to grasp (*red rectangles*) or to use/place them (*green rectangles*). Even though these activities are some of the most simple in BEHAVIOR-1K, they are still very long-horizon, requiring hundreds of low-level commands or the correct concatenation of multiple action primitives.

(a) navigate

(b) pick

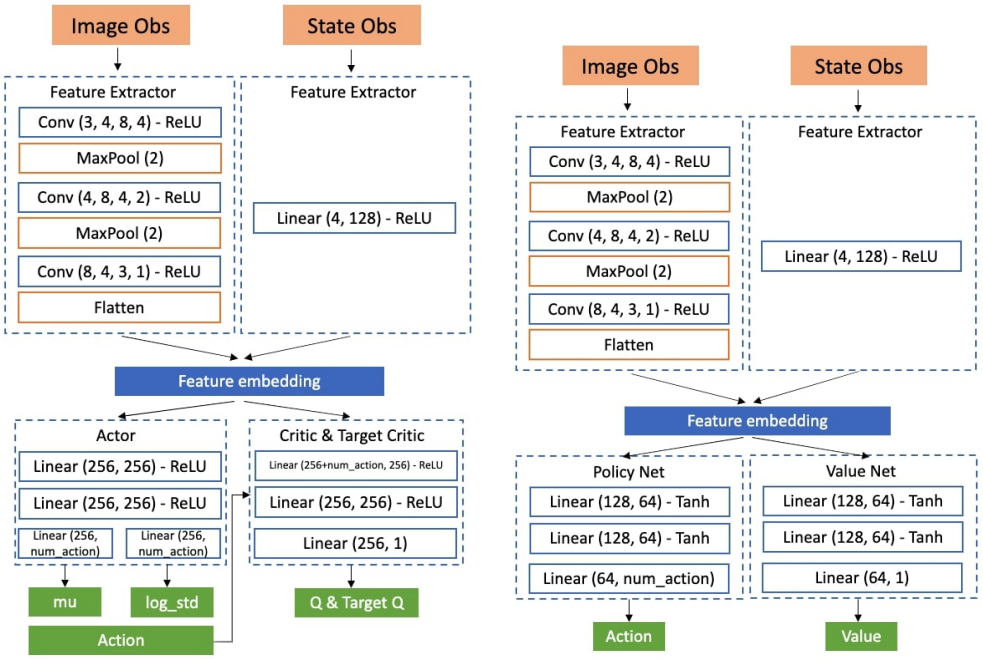
(c) place

(d) push (open)

(e) dip

(f) wipe

Figure A.10: Overview of all the high-level action primitives with the pre-conditions (left), post-conditions (right), and intermediate states (middle) during execution.



a) SAC Architecture b) PPO Architecture

Figure A.11: Policy architecture of SAC (RL-VMC) and PPO (RL-Prim. and RL-Prim.Hist.). The policy maps egocentric visual observations and proprioceptive information into an action that controls the robot base, arm, and gripper. PPO outputs a discrete high-level action primitive while SAC outputs a continuous low-level joint control directly.

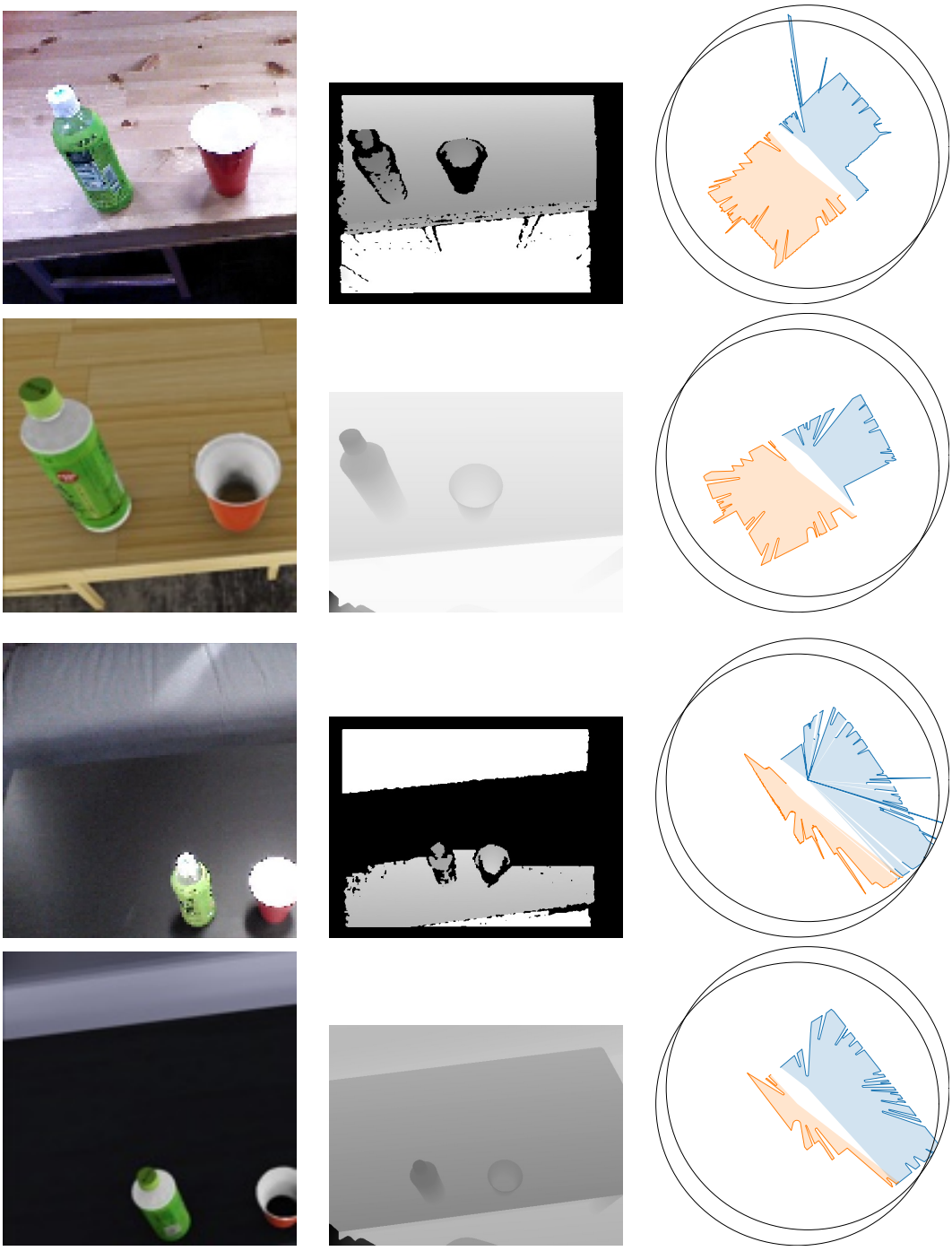


Figure A.12: **Comparison between real and simulated sensor signals.** RGB, Depth and LiDAR signals from the real-world sensors (first and third rows) and from the simulated sensors (second and fourth rows) at two different locations of the `mockup_apt` scene (location 1: first two rows, location 2: last two rows). We observed a smaller sensor sim-real gap for LiDAR than for RGB and Depth: RGB is heavily influenced by lighting conditions and camera settings while Depth has difficulty in capturing reflective surfaces.