

ANYTASK: an Automated Task and Data Generation Framework for Advancing Sim-to-Real Policy Learning

Ran Gong^{1*}, Xiaohan Zhang^{1*}, Jinghuan Shang^{1*}, Maria Vittoria Minniti^{1*}, Jigarkumar Patel¹, Valerio Pepe¹, Riedana Yan¹, Ahmet Gundogdu¹, Ivan Kapelyukh¹, Ali Abbas¹, Xiaoqiang Yan¹, Harsh Patel¹, Laura Herlant¹, Karl Schmeckpeper¹

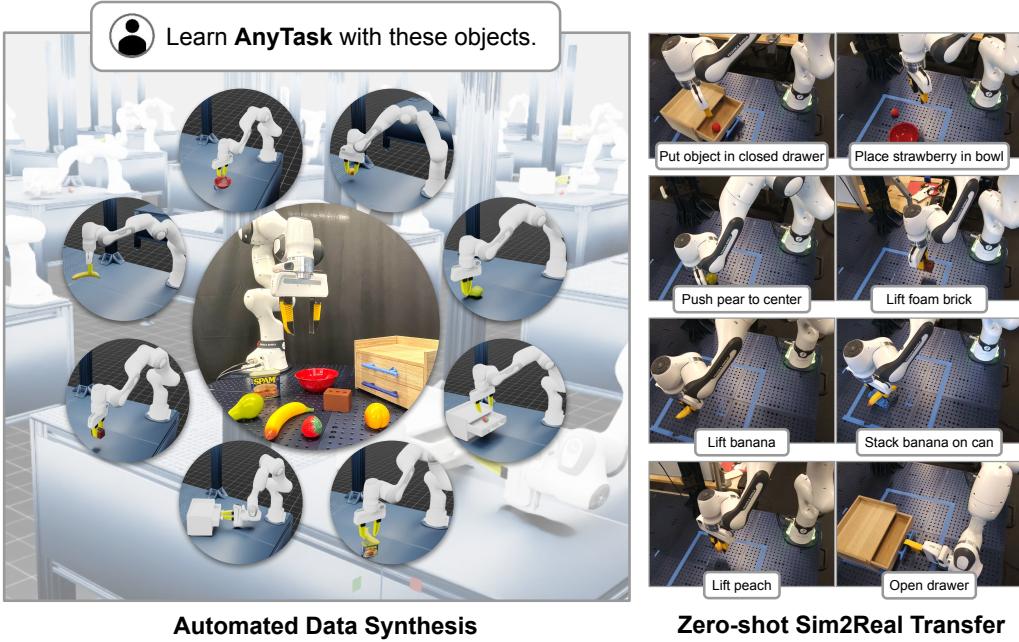


Fig. 1: ANYTASK is a framework that automates task design and generates data for robot learning. The resulting data enables training visuomotor policies that can be deployed directly onto a physical robot without requiring any real-world data.

Abstract—Generalist robot learning remains constrained by data: large-scale, diverse, and high-quality interaction data are expensive to collect in the real world. While simulation has become a promising way for scaling up data collection, the related tasks, including simulation task design, task-aware scene generation, expert demonstration synthesis, and sim-to-real transfer, still demand substantial human effort. We present ANYTASK, an automated framework that pairs massively parallel GPU simulation with foundation models to design diverse manipulation tasks and synthesize robot data. We introduce three ANYTASK agents for generating expert demonstrations aiming to solve as many tasks as possible: 1) ViPR, a novel task and motion planning agent with VLM-in-the-loop Parallel Refinement; 2) ViPR-EUREKA, a reinforcement learning agent

with generated dense rewards and LLM-guided contact sampling; 3) ViPR-RL, a hybrid planning and learning approach that jointly produces high-quality demonstrations with only sparse rewards. We train behavior cloning policies on generated data, validate them in simulation, and deploy them directly on real robot hardware. The policies generalize to novel object poses, achieving 44% average success across a suite of real-world pick-and-place, drawer opening, contact-rich pushing, and long-horizon manipulation tasks. Our project website is at <https://anytask.rai-inst.com>.

I. INTRODUCTION

The success of deep learning fundamentally depends on access to large-scale, high-quality data [1]–[3], as demonstrated in various domains such as language modeling [4]–[7], visual understanding [8]–[14], generation [15]–[17], and multimodal applications [18]–[20]. However, collecting robot data is extremely time-consuming and costly [21], [22] as it

* Equal Contribution

¹Robotics and AI Institute, Boston, MA, USA. {rgong, xzhang, jshang, mminniti, jpatel, vpepe, ryan, agundogdu, IKapelyukh, aabbas, xyan, hapatel, lherlant, kschmeckpeper}@rai-inst.com

necessitates direct physical interaction with the real world. Robot simulation, which can be scaled straightforwardly with compute [23]–[25], presents an appealing alternative for collecting large-scale datasets with minimal real-world effort [26]–[31]. While prior work has made significant progress in designing simulation systems for a wide range of tasks, tremendous human effort is often a huge barrier in building these systems [32], [33]. This effort includes proposing tasks, selecting task-relevant object assets, designing metrics, ensuring feasibility, and generating a large quantity of high-quality demonstration data. These non-trivial components frequently limit the diversity of the generated data.

Trained on vast internet data, foundation models demonstrate remarkable abilities in robotic downstream applications, such as scene understanding, task planning, motion synthesis, and low-level control [34]–[37]. These capabilities can also be leveraged to automate many key steps in creating robotic simulation environments, such as task design, writing simulation code, and iterative refinement. However, prior work leveraging foundation models for robot simulations either requires significant human efforts on task design and demonstration collection [30], [31], [38], or struggles with sim-to-real transfer [39], [40], even though the ultimate goal of large-scale data collection is to deploy the trained system in the real world.

To address the aforementioned challenges, we introduce ANYTASK (Figure 1), a scalable framework designed to bridge the gap between current simulators and a fully automated data generation system. The primary goal of ANYTASK is to leverage massively parallel GPU-based simulation engines and foundation models to generate high-quality, diverse scenes, tasks, and expert demonstrations at scale. To achieve this, our framework (as illustrated in Figure 2) integrates an intelligent object database, a task generator, and a simulation generator, all orchestrated by LLMs, to produce diverse, large-scale manipulation datasets for robust sim-to-real transfer (section III). To synthesize robotic trajectories for a diverse set of tasks, we introduce three ANYTASK agents built upon task and motion planning (TAMP) and reinforcement learning (RL): ViPR, ViPR-EUREKA, and ViPR-RL. This data is used to train visuomotor policies that are directly deployable in the real world (section IV). Furthermore, we design a task management workflow and a demonstration replay mechanism to accelerate the data collection process (section V). The entire pipeline, from task generation to policy training, operates almost autonomously, requiring only a high-level textual objective.

In summary, we make the following contributions:

- We present ANYTASK, a novel, automated framework that leverages massively parallel, GPU-based simulation to acquire robotic data from high-level goals, significantly reducing the need for manual intervention.
- Based on the highly parallel nature of our framework, we introduce ViPR, ViPR-EUREKA, and ViPR-RL agents that can automatically generate expert demonstrations on ANYTASK at scale.

- We validate the utility of our generated data by training and evaluating visuomotor policies on a suite of manipulation tasks in simulation.
- We demonstrate *zero-shot* policy transfer to a physical robot, and identify key factors, such as domain randomization and policy architecture, that are critical for bridging the sim-to-real gap.

II. RELATED WORKS

A. Large-Scale Robotics Dataset in Simulation

Recent progress in simulations enabled large-scale robot data collection [32], [45]–[47]; however, most tasks are still manually curated. This process requires substantial human effort to design, implement, and validate meaningful tasks. More recently, several studies have explored using large language models (LLMs) to automatically propose tasks [38], [40]–[43]. However, these efforts typically do not focus on scaling to larger datasets, addressing sim-to-real transfer, or developing diverse and systematic data collection strategies.

In contrast, we introduce a holistic, end-to-end pipeline designed to automate the entire data generation lifecycle while directly addressing the sim-to-real challenge, as shown in Table I. Our system integrates asset selection, scene configuration, task generation, task success criterion generation, policy data collection, policy distillation, and real-world deployment, all with significantly reduced human efforts.

B. Sim-to-Real Transfer

Recent years have witnessed impressive advancements in sim-to-real transfer [48]–[51]. However, these methods often rely on meticulously human-designed reward functions or complex training pipelines. A promising direction involves leveraging Large Language Models (LLMs); for instance, recent work has demonstrated the feasibility of using LLM-generated tasks for sim-to-real [38]. Our work demonstrates that by using LLMs to generate not only the tasks but also the data collection policies, we can achieve competitive sim-to-real performance. While concurrent work [31] also achieves zero-shot sim-to-real transfer, their approach relies on policies pre-trained with real-world robot data. In contrast, we establish that effective real-world policy transfer is achievable using a framework built upon purely synthetic data for a diverse range of tasks.

III. ANYTASK

ANYTASK aims to generate text-based task descriptions and corresponding runnable simulation code for agents to collect synthetic data. The system overview is available in Figure 2. In the sections below, we introduce Object Database, Task Generator, Simulation Generator, and other key infrastructure components.

a) Object Database: We build an object database storing objects' information so that retrieving objects through natural language is possible. The object database is built based on the available assets before task generation. The database encodes objects and object parts for their names, colors, textures, materials, bounding boxes (extent), joint

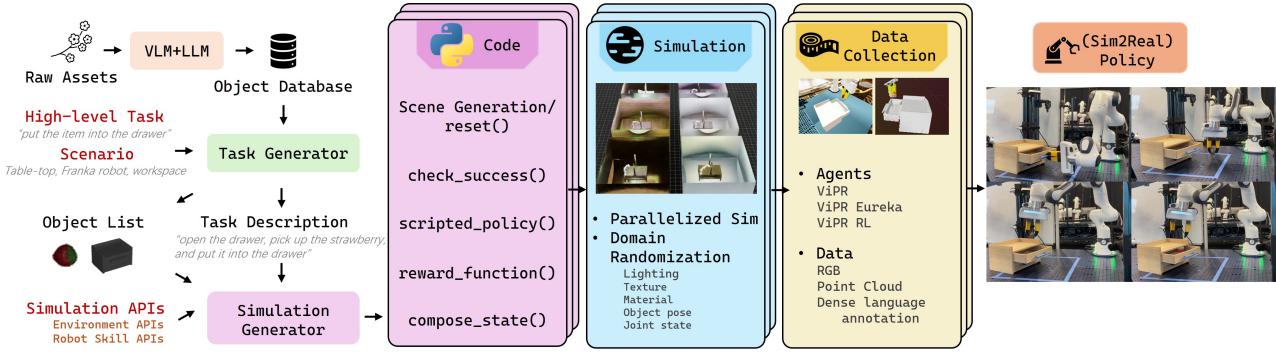


Fig. 2: Overview of ANYTASK. We first generate simulated manipulation tasks from an object database and a high-level task (i.e., task type). Then the pipeline automatically proposes task descriptions, generates the simulation code, and efficiently collects data using different agents, including ViPR, ViPR-RL, and ViPR-EUREKA in massively parallel simulation environments. We apply online domain randomization in the simulation to ensure the diversity of the scenes and the visual observations. Finally, we train the policy using simulated data and zero-shot transfer to the real world.

	Auto Task Generation	Auto Trajectory Generation	Auto Object	Dense Annotation	Task Metric Generation	Scene Generation	Domain Randomization	Massively Parallel GPU Simulation	Long Horizon	ZeroShot Perceptual Sim-to-real Transfer	Rendering
RoboGen [41]	✓		RL, TAMP, H	✓	✗	✓	✗	✗	✗	✗	R
GenSim [42]	✓		RL	✗	✓	✓	✗	✗	✗	✗	R
GenSim [40]	✓		TAMP	✗	✓	✓	✗	✗	✗	✗	R
GenSim2 [38]	✓		✗	✗	✓	✓	✗	✗	✗	✓	R
ScalingUp [43]	✓		TAMP	✗	✓	✓	✓	✗	✓	✓	R
RobotTwins [30], [31]	✗		✗	✗	✗	✗	✗	✗	✓	✓	R
Menasfield [29]	✗		RL	✗	✗	✗	✗	✗	✗	✗	R
Robocasa [44]	✗		J	✗	✗	✗	✗	✗	✗	✗	R
ARNOLD [27]	✗		TAMP	✗	✗	✓	✗	✗	✓	✓	RT
Manskill [32], [45]	✗		RL, TAMP	✗	✗	✗	✗	✗	✓	✓	RT
LIBERO [28]	✗		✗	✗	✗	✗	✗	✗	✓	✓	R
BehaviorIK [33]	✗		✗	✗	✗	✗	✗	✗	✓	✗	RT
AnyTask (ours)	✓		RL, TAMP, H	✓	✓	✓	✓	✓	✓	✓	RT

TABLE I: Comparison with other simulation systems. **Auto Task Generation:** Automatic task generation from a single text prompt. **Auto Trajectory Generation:** Automatic trajectory generation with no human effort. **RL:** Using reinforcement learning to generate demonstrations. **TAMP:** Using task and motion planning approach to generate demonstrations. **H:** Combing TAMP and RL in a single trajectory. **Auto Object:** Automatic object creation/indexing with no human effort. **Dense Annotation:** Dense annotation for each robot manipulation. **Task Metric Generation:** Task Metric Generation. **Scene Generation:** Automatic Scene Generation. **Domain Randomization:** Automatic Online Physical Visual Domain Randomization. **Massively Parallel GPU Simulation:** Massively parallel GPU-based simulation with massively parallel cameras to support scalable task, scene, and trajectory generation as well as domain randomization. **Long Horizon:** Long horizon task and demo generation. **ZeroShot Perceptual Sim-to-real Transfer:** Zero-shot Perceptual sim-to-real for a closed-loop policy. **RT:** Real-time ray tracing. **R:** Rasterization or non-real-time ray tracing.

information (for articulated objects), and overall descriptions. This process combines textual and visual information by rendering each object and part from multiple viewpoints, and asking a VLM (GPT-4o) to give the annotations related to visual properties. We use Sentence-T5-Large [52], [53] to compute sentence-level embeddings, and then use faiss [54], [55] to build an index for nearest neighbor search. Therefore, no human efforts are involved with object database creation and query, beyond finding assets. Examples of labeled object metadata can be found in appendix VII-B.1.

b) *High-level task and scenario information:* Our goal is to generate a diverse set of realistic and physically plausible robotic tasks. To achieve this, we prompt an LLM with high-level information, including a task family (e.g., “pick-and-place”), robot specifications, and workspace constraints, all provided in natural language by a human.

c) *Task Generator:* Task generator uses the high-level task and scenario information to propose tasks and objects with the help of the object database. We support two variants. In **object-based task generation**, objects are first sampled from a database, and the LLM then proposes a detailed task involving them. This is flexible for general tasks like “pick-

and-place”. In **task-based object proposal**, the LLM first suggests objects suitable for a given task (e.g., a drawer for “open a drawer”). The system then retrieves a matching asset from the object database, and the LLM generates the final, detailed task description. This stage outputs a natural language task description and structured object metadata, including the object details in the object database.

d) *Simulation Generator:* Our simulation generator takes the task description and objects as input and produces code that can execute based on our simulation framework. To execute the generated code and leverage massively GPU simulations, we use IsaacLab [24] simulator. We choose to generate code to define a task since code has less ambiguity than natural language and has higher flexibility than configuration files. We provide environment and robot skill API definitions as part of the prompt. The APIs are designed by humans to ensure correctness. API lists are available in Table XI and Table XII.

In detail, the LLM is required to generate the code for five key functions: `reset()` for randomizing the scene (e.g., object poses), `check_success()` to define the task’s goal

condition, `compose_state()` to provide a state representation for an RL policy, `reward_function()` for an initial version of reward function in RL, and `scripted_policy()` to define an expert policy for data collection. To ensure these functions are consistent, we generate `check_success()` first and use it to instruct the LLM when generating the other four functions.

e) Dense Annotation: Language annotations are limited in existing robot datasets [21], [22], where only one or a few sentences of task description are often paired with one demonstration. We introduce our automated dense annotation system to bridge this gap. We transform the privileged information in the simulation into dense, natural language annotations to summarize the environmental states before and after executing an action. Each annotation is tagged to a certain timestep or a period of time of the trajectory. To generate the annotations, we provide an API `log_step()` so that the LLM can call it any time during policy execution, and decide what information to include using other environment APIs. In this way, we can automatically generate data with rich text annotations, providing strong support for our policy refinement (introduced later). An example dense annotation is shown below, where the variables will be replaced by the actual value in the simulation.

```
{
  'step': 0,
  'content': {
    'step_description': {'action': 'Move end-effector to drawer handle', ...},
    'object_states': {'drawer_handle': {'position': [x,y,z], ...}, ...},
    'robot_state': {'eef_pos': [x,y,z], ...}
  },
  ...
}
```

A more integrated example can be found in Figure 19.

IV. ANYTASK AGENTS

This section describes the agents we developed and evaluated for ANYTASK. Our guiding principle is to *explore how a robot agent can solve as many generated tasks as possible with no human effort*.

In this work, we study task and motion planning (TAMP) and reinforcement learning (RL), two commonly used teacher policies in manipulation tasks. TAMP is known for handling long-horizon tasks, while RL excels at dexterous, contact-rich manipulation. However, traditional TAMP methods require pre-defined domain and action knowledge, usually specified in PDDL format by domain experts. RL, on the other hand, is typically limited to specific domains where practitioners can carefully construct reward functions that provide accurate learning signals for the desired behavior.

To this end, we introduce three ANYTASK agents for generating expert demonstrations:

- ViPR, a novel TAMP agent with VLM-in-the-loop Parallel Refinement,
- ViPR-EUREKA, an improved version of Eureka [56] with VLM-finetuned sparse rewards and Mesh-based Contact Sampling, and
- ViPR-RL, a hybrid TAMP+RL approach.

A. ViPR

A ViPR agent uses an LLM to produce a task-motion plan p as a Python program that calls our parameterized skill APIs (in our case, `move_to`, `open_gripper`, `close_gripper`, `grasp`), following prior approaches on code generation for robot control [57]. Naively running the generated programs in an open-loop manner often leads to failure, mainly because LLMs (and foundation models in general) lack robust spatial understanding of the environment [58]. A common failure mode is commanding inaccurate 3D positions or orientations for the end-effector.

To mitigate this limitation, we propose to use VLMs for iteratively refining the task-motion plan. Each refinement iteration takes as input: the current plan p , images collected during rollout, dense annotations from ANYTASK, and the available skill and environment APIs. The iteration outputs an updated plan p' . We execute K parallel rollouts of p in simulation to (i) record videos and dense trajectory annotations and (ii) expose diverse failure modes in a single pass. A VLM evaluates every such rollout and outputs natural-language feedback plus a binary success/failure with confidence. We aggregate these per-episode judgments into a scalar which is the success rate across the K rollouts combined with average confidence and compare VLM judgments against `check_success` that only inspects initial and final states to monitor agreement. A generated example can be found in Figure 19.

B. ViPR-EUREKA

To generate demonstrations with reinforcement learning, we use an improved version of Eureka [56] to iteratively refine and sample reward functions proposed by LLM.

Mesh-based Contact Sampling: A core component of our approach is a novel contact sampling algorithm. The sampler generates high-quality grasp candidates by first sampling a triangle on the object mesh and applying barycentric interpolation [59] to determine a contact position.

The object of interest is first generated by an LLM. The right gripper finger is then positioned along the perturbed surface normal at the sampled location with Gaussian noise. To speed up sampling efficiency, the gripper orientation is randomly sampled around a predefined gripper z-axis produced by a vision language model (VLM) or human users.

To ensure feasibility, we employ a rejection sampling mechanism that discards grasp candidates with collisions or invalid orientations based on collision checking. We sample and check around 1024 candidates per environment in parallel using batched collision checking and batched inverse kinematics (IK) with cuRobo [60]. The resulting valid position is then used as the initial state for RL training. After the training success rate improves, we gradually decay the number of environments using contact sampling. Some example contact sampling results are shown in Figure 25 and an example ViPR-RL policy can be found in Figure 20.

C. ViPR-RL

We want to combine the strengths of both worlds, RL and motion planning, since RL is good with contact-rich tasks, and motion planning is good at free space movement. Several modifications are needed: 1) The code generation now includes trained RL skills with APIs. 2) For each sub-task, we use motion planning to move the gripper to the object parts of interest, which are sampled by the grasp sampler described above, and then we invoke trained RL skills. To train an object-based RL skill, we run the PPO 1500 epochs with 1024 environments for each object; it typically requires 20 minutes on an L4 GPU for a single object. The reward function is a simple success checker produced by LLM. A sample code snippet is shown below.

```
def ViPR_policy_rl(env):
    # Object IDs
    baseball_id = 1
    # Get the grasp pose from the RL skill API
    grasp_position, grasp_orientation = get_grasp_position(
        env, baseball_id, part_name="")
    )
    # Move to the grasp pose
    hover_offset = torch.tensor([[-0.0, 0.0, 0.1]])
    above_grasp = grasp_position + hover_offset
    move_to(env=env, target_position=above_grasp,
            target_orientation=grasp_orientation, gripper_open=True)
    move_to(env=env, target_position=grasp_position,
            target_orientation=grasp_orientation, gripper_open=True)
    # Execute the RL picking skill
    pick_success = pick_rl(env, external_id=baseball_id)
    move_to(env=env, target_position=above_grasp,
            target_orientation=None, gripper_open=False)
```

A more integrated example can be found in Figure 21 in the appendices.

V. INFRASTRUCTURE DESIGN

A. Multi-GPU Data Collection

The data collection pipeline is orchestrated using Metaflow [61] to manage the sequential execution of each stage and the data artifacts produced within a simulation environment. There are three stages. It begins with an optional policy refinement stage, allowing for the enhancement of an agent’s performance prior to data collection. In the second stage, the primary data gathering is conducted using a state-based policy, which efficiently captures a diverse range of interaction trajectories without rendering. In the final stage, these collected trajectories are replayed to render and capture high-fidelity vision data. This decoupling of collection logic from the rendering process significantly reduces computational overhead and allows for independent iteration on visual parameters. We launch a Metaflow pipeline on each GPU node so data can be collected in parallel. The resulting Metaflow-managed agents provide a fast and adaptable workflow for generating large-scale, vision-based datasets.

B. Demonstration Replay

Instead of generating and recording demonstrations at the same time, we first execute a state-based policy, (ViPR, ViPR-EUREKA or ViPR-RL) to generate numerous rollouts in parallel without recording. We then store the states from

only the successful trajectories. These successful trajectories are replayed to train our final policy. We employ two replay methods: 1) State Replay: We directly set the simulation to the stored states of a successful trajectory. 2) Action Replay: We re-execute the original action sequence from a successful trajectory.

VI. EXPERIMENTS

We perform experiments to evaluate our code generation (VI-A) and the diversity (VI-B) of our tasks, the success rates (VI-C) and speed (VI-D) of data generation, and the performance of policies trained on our data in simulation (VI-E) and the real world (VI-F).

A. Are programs synthesized by ANYTASK runnable in simulation?

TABLE II: Code runability analysis.

LLM	Runnable	Error Type		
		compose_state()	reset()	check_success()
o1-mini	64%	0%	20%	16%
DeepSeek-R1 [62]	76%	0%	16%	8%
o3-mini	84%	4%	12%	0%
o3-mini + improved prompt	96%	4%	0%	0%

ANYTASK relies on LLM to generate code that runs in the simulation. We tested several LLMs: o1-mini, DeepSeek-R1-671B [62], and o3-mini by using them to generate 20 tasks with the same set of objects. The test only focuses on the basic simulation environment loop, not the policy, so only `compose_state()`, `reset()`, and `check_success()` are executed. We report the code runability – the ratio of the code that can run in simulation, and in which functions errors may frequently occur. Table II reports the code runability. We find that o3-mini gives the highest code runability. We also find that the errors often come from the `reset()` function, since that function requires strong logic to handle object placement and spatial transforms. We further summarize the errors in these tests and compose an improved prompt targeted to those errors. With the improved prompt, ANYTASK can achieve a code runability of 96% using o3-mini.

B. How diverse is ANYTASK compared to other data generation systems in the literature?

TABLE III: BLEU Score of Generated Task Descriptions

Ours	RoboGen [41]	RLBench [26]	Gensim2 [38]
0.352	0.494	0.590	0.692

Diversity is one of the key aspects of data quality. We use self-BLEU score [63] to evaluate the diversity of the generated task descriptions. We compare our system against RoboGen [41], RLBench [26], and GenSim2 [38]. Since our system requires human input for high-level tasks, we use the high-level manipulation tasks from RoboGen [41] to generate our detailed task descriptions. We compute the self-BLEU score of the task descriptions from each method using `n_grams=4`. The result is available in Table III. Our

system has the lowest self-BLEU score, showing that our task descriptions have better diversity than other methods.

C. How robust are ANYTASK agents on generating expert demonstrations?

We collect data using ViPR across five task categories: **lifting**, **pick-and-place**, **pushing**, **stacking**, and **drawer opening** with varying difficulties, totaling more than 400 tasks.

As shown in Table IV, each agent excels at different tasks, enabling the ensemble to collectively solve more tasks than any single agent. This highlights the necessity of agent diversity, as certain approaches are ill-suited for specific agents. Qualitatively, ViPR-EUREKA is able to learn to grasp a grasping complex object, like a bleach bottle, while the other methods fail because they cannot explore enough to find the single viable angle. ViPR-RL can solve a stacking task that requires knocking over one of the objects before stacking the second object on top, while ViPR cannot learn to knock over the object and ViPR-EUREKA struggles with the multi-step nature of the task. Finally, ViPR is most successful at multi-step tasks that do not require unique behaviors.

TABLE IV: Percentage of tasks that ANYTASK agents can successfully solve (i.e., success rate >10%).

	Lifting	Pushing	Stacking	Pick & Place	Drawer opening
ViPR w/o Refinement	58%	30%	26%	66%	0%
ViPR	81%	54%	44%	76%	0%
ViPR-RL	35%	-%	9%	33%	33%
ViPR-EUREKA	69%	60%	33%	73%	17%
All	90%	70%	54%	87%	33%

1) *Is refinement in ViPR useful?:* The VLM refiner improves success rates in 86.4% of tasks, with an average gain of 13.6% for tasks with non-zero initial success. This consistently produces more robust policies Figure 3.

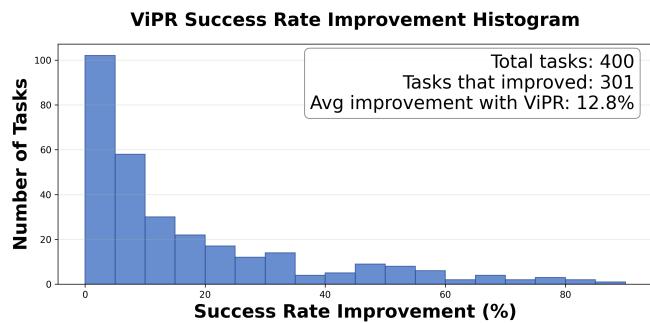


Fig. 3: ViPR improvement: Using ViPR leads to an average 12.8% improvement in success rate on 301 tasks

2) *Is contact sampling useful?:* To demonstrate the effectiveness of LLM-guided contact sampling, we perform ablation studies against the vanilla Eureka.

As shown in Table V, ViPR-EUREKA significantly outperforms the standard Eureka. All experiments are run across **30** tasks within the task family, with 3 Eureka iterations and 3 tries each. A task was considered successful if any iteration in any try achieved a success rate exceeding 10% .

TABLE V: Data collection RL policy training success rate with and without contact sampling

	Lifting	Pushing	Stacking	Pick&Place	DrawerOpening	Avg.
Eureka	40 %	40 %	0 %	57 %	50 %	37 %
ViPR-EUREKA	73 %	50 %	57 %	87 %	17 %	57 %

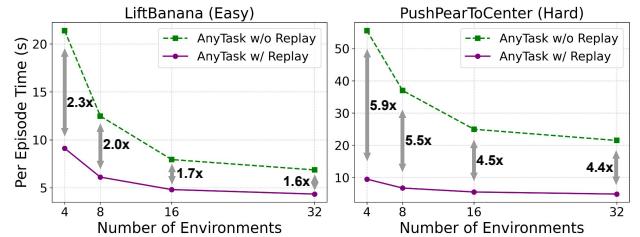


Fig. 4: Action replay enables faster data collection, especially on challenging tasks.

D. How fast can ANYTASK agents generate data?

The throughput of ANYTASK data generation is determined by two key factors: the success rate of the AnyTask Agents and the trajectory length (i.e., the number of simulation timesteps) required to complete a task. To optimize throughput, we decompose the pipeline into two stages: (1) demonstration recording and (2) trajectory replay. During the first stage, AnyTask agents attempt tasks without rendering, and only simulator states from successful trajectories are stored. In the second stage, the saved simulator states are replayed with rendering enabled to generate the full dataset, including RGB images, colored point clouds, robot states, and action sequences required for imitation learning. Our two-stage recording pipeline is highly efficient. In a single **~36 minutes** session on an L4 GPU, we collected 500 demonstrations, recording RGB-D and point cloud data from 4 cameras for each 11-second demo. This total time accounts for all overhead, including instance launching, isaac-sim shader compiling, point cloud computation, data saving and data uploading time.

As shown in Figure 4, action replay significantly improves data generation throughput by eliminating wasted rendering time. This benefit is especially pronounced on more difficult tasks, where agents often struggle to generate successful trajectories. In 4-camera environments, this method resulted in a four-fold speedup on more difficult tasks.

E. Can we train BC policies with data generated by ANYTASK agents?

TABLE VI: Policy evaluations in simulation.

Task family	ViPR	ViPR-RL	ViPR-EUREKA
Lifting	42.0%	9.4%	4.7%
Pick and place	40.7%	0.8%	-
Pushing	29.3%	-	19.8%
Stacking*	2.0%	0%	1.7%
Drawer	-	29.1%	2.6%

We train diffusion policies on each of the generated tasks. Table VI shows the policy success rates in simulation on a subset of the tasks that all data collection methods

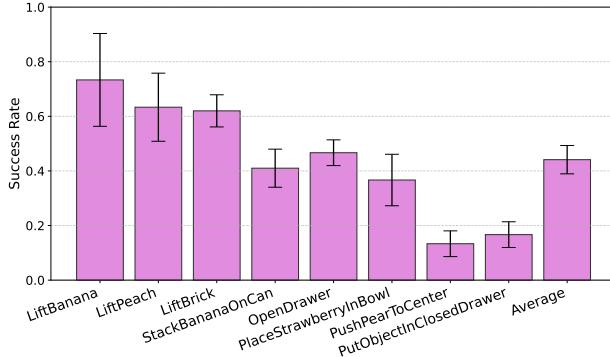


Fig. 5: Zero-shot sim-to-real policy evaluations.

successfully generated data for, comparing over 70 policies. ViPR data performs better on tasks that involve long horizon or multi-step processes, while data collection methods that include RL perform equal or better on tasks that involve continuous contact. Despite high data collection efficiencies, ViPR-EUREKA data is more difficult to distill into BC policies. For pick-and-place tasks, it hacks the reward system by pushing objects instead of picking them up. Example task descriptions that policies were successful on are shown in Table VII.

TABLE VII: Example task descriptions

Grasp the strawberry and lift it vertically off the table by about 10 cm, then hold it steady for a few seconds while ensuring the nearby plate remains undisturbed.
 Pick up the extra large clamp and place it slightly forward (positive x direction) relative to the cup.
 Push the pear diagonally (forward and left) so that it settles between the racquetball and the fork.
 Stack the baseball on top of the potted meat can, ensuring the baseball is directly aligned above the can.

More details related to policy learning are in the appendix VII-D.

F. Are these policies transferrable to real world?

We generated eight tasks (see Figure 5) in ANYTASK and used ViPR to collect 1,000 expert demonstrations per task. These demonstrations vary in length from 10 seconds (e.g., *LiftBanana*) to 30 seconds (e.g., *PutObjectInClosedDrawer*, where the robot opens the drawer, picks up the strawberry, and places it inside). We distill these demonstrations into a set of single-task, point-cloud-based policies using 3D Diffusion Policy [64]. Note that for drawer-related tasks, we provide an additional *open_drawer* skill API for ViPR to generate high-quality trajectories. Each single-task policy is trained on $4 \times$ NVIDIA H100 GPUs for 500 epochs with a global batch size of 1,024. We use a cosine learning-rate schedule (initial LR 5×10^{-5}) with 100 warm-up iterations and weight decay 1×10^{-6} .

For better sim-to-real performance, we use uncolored point clouds as visual input. The workspace uses four tabletop RealSense D455 cameras, and we use an image resolution of 320×240 . We fuse points from the four cameras and uniformly subsample to 4,096 points. We crop out table points. We apply small pose jitter: translations in $[-1, 1]$ cm and rotations in $[-2^\circ, 2^\circ]$. To mimic depth artifacts, we simulate “ghost” points: up to 5% of the cloud, 70% biased

near object boundaries within a 10% shell; depths include Gaussian noise with $\sigma = 3$ mm. We use the absolute end-effector pose and a discrete gripper state (0=open, 1=closed) as proprioceptive inputs.

For the action space, the policy predicts a chunk of 64 actions and each action is an absolute end-effector poses and a desired gripper state. We implement an asynchronous policy runner with temporal ensembling [65] to execute 32 actions from each predicted chunk. The policy runs locally on a single A6000 GPU at 30Hz.

We evaluate each of the eight policies for 30 trials with randomly sampled object poses within the workspace. Error bars are computed by partitioning the 30 trials into three groups of 10 and reporting the mean \pm s.e.m. across groups. Figure 5 shows per-task success; the policies show generalization to novel object poses, achieving 44% average success. More details can be found in appendix VII-E.

VII. CONCLUSIONS, LIMITATIONS AND FUTURE WORK

In this work, we addressed the critical data bottleneck in robot learning by introducing ANYTASK, a framework that automates the entire pipeline from high-level task to sim-to-real policy deployment. We demonstrated how ANYTASK leverages foundation models and parallel simulation to automatically generate diverse tasks, scenes, and success criteria. Our novel data generation agents, including the TAMP-based ViPR and RL-based ViPR-EUREKA and ViPR-RL, efficiently produce high-quality expert demonstrations for a wide range of manipulation challenges. Our approach is validated by training a visuomotor policy purely on this synthetic data and deploying it zero-shot to a physical robot, achieving notable performance across various tasks without any real-world fine-tuning.

Despite these promising results, our framework has several limitations that present exciting avenues for future research. First, while our agents demonstrate broad capabilities, their performance varies on tasks requiring high-precision or complex physical reasoning, such as stacking arbitrary objects. Second, our successful sim-to-real transfer relied on point-cloud observations. Extending this to RGB-based policies would be a valuable direction, as it would lower the barrier for real-world deployment on a wider variety of hardware. We also plan to scale the framework to include a greater diversity of objects and robot morphologies, as well as extend it to more complex, long-horizon mobile manipulation tasks.

REFERENCES

- [1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.
- [2] S. Toshniwal, W. Du, I. Moshkov, B. Kisacanin, A. Ayrapetyan, and I. Gitman, “Openmathinstruct-2: Accelerating ai for math with massive open-source instruction data,” *arXiv preprint arXiv:2410.01560*, 2024.
- [3] C. Schuhmann, R. Beaumont, R. Vencu, C. Gordon, R. Wightman, M. Cherti, T. Coombes, A. Katta, C. Mullis, M. Wortsman, *et al.*, “Laion-5b: An open large-scale dataset for training next generation image-text models,” *Advances in neural information processing systems*, vol. 35, pp. 25278–25294, 2022.

- [4] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, *et al.*, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [5] H. Touvron *et al.*, “Llama: Open and efficient foundation language models,” *arXiv preprint arXiv:2302.13971*, 2023.
- [6] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, *et al.*, “Deepseek-v3 technical report,” *arXiv preprint arXiv:2412.19437*, 2024.
- [7] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, *et al.*, “Qwen technical report,” *arXiv preprint arXiv:2309.16609*, 2023.
- [8] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.
- [9] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, “An image is worth 16x16 words: Transformers for image recognition at scale,” in *International Conference on Learning Representations*, 2021.
- [10] M. Tschannen *et al.*, “Siglip 2: Multilingual vision-language encoders with improved semantic understanding, localization, and dense features,” *arXiv preprint arXiv:2502.14786*, 2025.
- [11] J. Shang, K. Schmeckpeper, B. B. May, M. V. Minniti, T. Kelestemur, D. Watkins, and L. Herlant, “Theia: Distilling diverse vision foundation models for robot learning,” *arXiv preprint arXiv:2407.20179*, 2024.
- [12] G. Heinrich, M. Ranzinger, H. Yin, Y. Lu, J. Kautz, A. Tao, B. Catanzaro, and P. Molchanov, “Radiov2.5: Improved baselines for agglomerative vision foundation models,” in *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 22487–22497, 2025.
- [13] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, *et al.*, “Learning transferable visual models from natural language supervision,” in *International Conference on Machine Learning*, pp. 8748–8763, PMLR, 2021.
- [14] J.-B. Alayrac, J. Donahue, P. Luc, A. Miech, I. Barr, Y. Hasson, K. Lenc, A. Mensch, K. Millican, M. Reynolds, *et al.*, “Flamingo: a visual language model for few-shot learning,” *Advances in neural information processing systems*, vol. 35, pp. 23716–23736, 2022.
- [15] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, “High-resolution image synthesis with latent diffusion models,” 2021.
- [16] H. Li, H. Shi, W. Zhang, W. Wu, Y. Liao, L. Wang, L.-h. Lee, and P. Y. Zhou, “Dreamscene: 3d gaussian-based text-to-3d scene generation via formation pattern sampling,” in *European Conference on Computer Vision*, pp. 214–230, Springer, 2024.
- [17] Z. Chen, G. Wang, and Z. Liu, “Scenedreamer: Unbounded 3d scene generation from 2d image collections,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 45, no. 12, pp. 15562–15576, 2023.
- [18] H. Liu, C. Li, Q. Wu, and Y. J. Lee, “Visual instruction tuning,” *Advances in neural information processing systems*, vol. 36, pp. 34892–34916, 2023.
- [19] M. Shridhar, L. Manuelli, and D. Fox, “Cliport: What and where pathways for robotic manipulation,” in *Conference on robot learning*, pp. 894–906, PMLR, 2022.
- [20] X. Li, C. Mata, J. Park, K. Kahatapitiya, Y. S. Jang, J. Shang, K. Ranasinghe, R. Burgert, M. Cai, Y. J. Lee, *et al.*, “Llara: Supercharging robot learning data for vision-language policy,” *arXiv preprint arXiv:2406.20095*, 2024.
- [21] K. Black, N. Brown, D. Driess, A. Esmail, M. Equi, C. Finn, N. Fusai, L. Groom, K. Hausman, B. Ichter, *et al.*, “pi0: A vision-language-action flow model for general robot control,” *arXiv preprint arXiv:2410.24164*, 2024.
- [22] G. Team, “Galaxea g0: Open-world dataset and dual-system vla model,” *arXiv preprint arXiv:2509.00576v1*, 2025.
- [23] NVIDIA, “Isaac Sim.”
- [24] M. Mittal *et al.*, “Orbit: A unified simulation framework for interactive robot learning environments,” *IEEE Robotics and Automation Letters*, 2023.
- [25] F. Xiang, Y. Qin, K. Mo, Y. Xia, H. Zhu, F. Liu, M. Liu, H. Jiang, Y. Yuan, H. Wang, *et al.*, “Sapien: A simulated part-based interactive environment,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 11097–11107, 2020.
- [26] S. James, Z. Ma, D. Rovick Arrojo, and A. J. Davison, “Rlbench: The robot learning benchmark & learning environment,” *IEEE Robotics and Automation Letters*, 2020.
- [27] R. Gong, J. Huang, Y. Zhao, H. Geng, X. Gao, Q. Wu, W. Ai, Z. Zhou, D. Terzopoulos, S.-C. Zhu, B. Jia, and S. Huang, “Arnold: A benchmark for language-grounded task learning with continuous states in realistic 3d scenes,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 20483–20495, October 2023.
- [28] B. Liu, Y. Zhu, C. Gao, Y. Feng, Q. Liu, Y. Zhu, and P. Stone, “Libero: Benchmarking knowledge transfer for lifelong robot learning,” *Advances in Neural Information Processing Systems*, 2023.
- [29] T. Yu, D. Quillen, Z. He, R. Julian, K. Hausman, C. Finn, and S. Levine, “Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning,” in *Conference on robot learning*, PMLR, 2020.
- [30] Y. Mu, T. Chen, Z. Chen, S. Peng, Z. Lan, Z. Gao, Z. Liang, Q. Yu, Y. Zou, M. Xu, *et al.*, “Robotwin: Dual-arm robot benchmark with generative digital twins,” in *Proceedings of the Computer Vision and Pattern Recognition Conference*, 2025.
- [31] T. Chen, Z. Chen, B. Chen, Z. Cai, Y. Liu, Q. Liang, Z. Li, X. Lin, Y. Ge, Z. Gu, *et al.*, “Robotwin 2.0: A scalable data generator and benchmark with strong domain randomization for robust bimanual robotic manipulation,” *arXiv preprint arXiv:2506.18088*, 2025.
- [32] S. Tao, F. Xiang, A. Shukla, Y. Qin, X. Hinrichsen, X. Yuan, C. Bao, X. Lin, Y. Liu, and T. kai Chan *et al.*, “Maniskill3: Gpu parallelized robotics simulation and rendering for generalizable embodied ai,” *Robotics: Science and Systems*, 2025.
- [33] C. Li *et al.*, “Behavior-1k: A human-centered, embodied ai benchmark with 1,000 everyday activities and realistic simulation,” *arXiv preprint arXiv:2403.09227*, 2024.
- [34] M. J. e. a. Kim, “Openvla: An open-source vision-language-action model,” in *Proceedings of The 8th Conference on Robot Learning*, 2025.
- [35] B. Liu, Y. Jiang, X. Zhang, Q. Liu, S. Zhang, J. Biswas, and P. Stone, “Llm+ p: Empowering large language models with optimal planning proficiency,” *arXiv preprint arXiv:2304.11477*, 2023.
- [36] Q. Gu *et al.*, “Conceptgraphs: Open-vocabulary 3d scene graphs for perception and planning,” in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5021–5028, IEEE, 2024.
- [37] J. Cui, T. Liu, N. Liu, Y. Yang, Y. Zhu, and S. Huang, “Anyskill: Learning open-vocabulary physical skill for interactive agents,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 852–862, 2024.
- [38] P. Hua, M. Liu, A. Macaluso, Y. Lin, W. Zhang, H. Xu, and L. Wang, “Gensim2: Scaling robot data generation with multi-modal and reasoning llms,” in *8th Annual Conference on Robot Learning*, 2024.
- [39] Y. Wang, Z. Xian, F. Chen, T.-H. Wang, Y. Wang, K. Fragkiadaki, Z. Erickson, D. Held, and C. Gan, “Robogen: Towards unleashing infinite data for automated robot learning via generative simulation,” *arXiv preprint arXiv:2311.01455*, 2023.
- [40] L. Wang, Y. Ling, Z. Yuan, M. Shridhar, C. Bao, Y. Qin, B. Wang, H. Xu, and X. Wang, “Gensim: Generating robotic simulation tasks via large language models,” *arXiv preprint arXiv:2310.01361*, 2023.
- [41] Y. Wang, Z. Xian, F. Chen, T.-H. Wang, Y. Wang, K. Fragkiadaki, Z. Erickson, D. Held, and C. Gan, “Robogen: Towards unleashing infinite data for automated robot learning via generative simulation,” 2023.
- [42] P. Katara, Z. Xian, and K. Fragkiadaki, “Gen2sim: Scaling up robot learning in simulation with generative models,” in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2024.
- [43] H. Ha, P. Florence, and S. Song, “Scaling up and distilling down: Language-guided robot skill acquisition,” in *Conference on Robot Learning*, PMLR, 2023.
- [44] S. Nasiriany, A. Maddukuri, L. Zhang, A. Parikh, A. Lo, A. Joshi, A. Mandlekar, and Y. Zhu, “Robocasa: Large-scale simulation of everyday tasks for generalist robots,” in *Robotics: Science and Systems*, 2024.
- [45] J. Gu, F. Xiang, X. Li, Z. Ling, X. Liu, T. Mu, Y. Tang, S. Tao, X. Wei, Y. Yao, X. Yuan, P. Xie, Z. Huang, R. Chen, and H. Su, “Maniskill2: A unified benchmark for generalizable manipulation skills,” in *International Conference on Learning Representations*, 2023.
- [46] S. Deng, M. Yan, S. Wei, H. Ma, Y. Yang, J. Chen, Z. Zhang, T. Yang, X. Zhang, H. Cui, *et al.*, “Graspvla: a grasping foundation model pre-trained on billion-scale synthetic action data,” *arXiv preprint arXiv:2505.03233*, 2025.

- [47] O. Mees, L. Hermann, E. Rosete-Beas, and W. Burgard, “Calvin: A benchmark for language-conditioned policy learning for long-horizon robot manipulation tasks,” *IEEE Robotics and Automation Letters (RA-L)*, 2022.
- [48] B. Tang, M. A. Lin, I. Akinola, A. Handa, G. S. Sukhatme, F. Ramos, D. Fox, and Y. Narang, “Industreal: Transferring contact-rich assembly tasks from simulation to reality,” *arXiv preprint arXiv:2305.17110*, 2023.
- [49] A. Handa, A. Allshire, V. Makoviychuk, A. Petrenko, R. Singh, J. Liu, D. Makoviichuk, K. Van Wyk, A. Zhurkevich, B. Sundaralingam, *et al.*, “Dextreme: Transfer of agile in-hand manipulation from simulation to reality,” in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023.
- [50] I. Akkaya *et al.*, “Solving rubik’s cube with a robot hand,” *arXiv preprint arXiv:1910.07113*, 2019.
- [51] A. Yu, A. Foote, R. Mooney, and R. Martín-Martín, “Natural language can help bridge the sim2real gap,” *arXiv preprint arXiv:2405.10020*, 2024.
- [52] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *Journal of machine learning research*, vol. 21, no. 140, pp. 1–67, 2020.
- [53] N. Reimers and I. Gurevych, “Sentence-bert: Sentence embeddings using siamese bert-networks,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, 11 2019.
- [54] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvassy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, “The faiss library,” 2024.
- [55] J. Johnson, M. Douze, and H. Jégou, “Billion-scale similarity search with GPUs,” *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.
- [56] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, L. Fan, and A. Anandkumar, “Eureka: Human-level reward design via coding large language models,” in *ICRL*, 2024.
- [57] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, “Code as policies: Language model programs for embodied control,” *arXiv preprint arXiv:2209.07753*, 2022.
- [58] A. Majumdar *et al.*, “Openeqa: Embodied question answering in the era of foundation models,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2024.
- [59] R. Osada, T. Funkhouser, B. Chazelle, and D. Dobkin, “Shape distributions,” *ACM Transactions on Graphics*, 2002.
- [60] B. Sundaralingam, S. K. S. Hari, A. Fishman, C. Garrett, K. Van Wyk, V. Blukis, A. Millane, H. Oleynikova, A. Handa, F. Ramos, *et al.*, “curobo: Parallelized collision-free minimum-jerk robot motion generation,” *arXiv preprint arXiv:2310.17274*, 2023.
- [61] N. O. S. Platform, “Metaflow.” <https://github.com/Netflix/metaflow>, 2019.
- [62] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, *et al.*, “Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning,” *arXiv preprint arXiv:2501.12948*, 2025.
- [63] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.
- [64] Y. Ze, G. Zhang, K. Zhang, C. Hu, M. Wang, and H. Xu, “3d diffusion policy: Generalizable visuomotor policy learning via simple 3d representations,” *arXiv preprint arXiv:2403.03954*, 2024.
- [65] T. Z. Zhao, V. Kumar, S. Levine, and C. Finn, “Learning fine-grained bimanual manipulation with low-cost hardware,” *arXiv preprint arXiv:2304.13705*, 2023.
- [66] D. P. Kingma, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

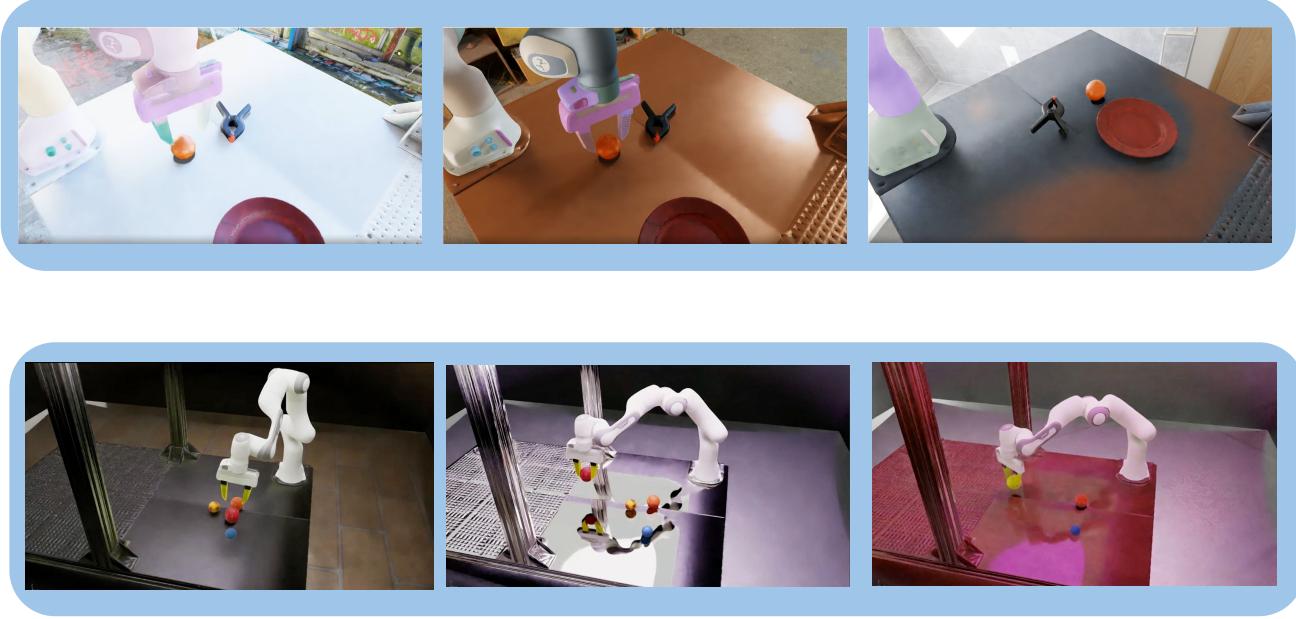


Fig. 6: Our system can automatically randomize object poses and textures

APPENDICES

A. Simulation Environments

1) *Rendering*: We demonstrate domain randomization capabilities of our system in Figure 6.

2) *Assets*: We display part of our simulated assets and real world assets in Figure 7. For the drawer, we manually create the URDF file and its associated links after scanning with an iPhone.

B. Task Generation

1) *Object Database*: We build the object database with VLM. Figure 8 and Figure 9 shows examples of our multi-view, multi-part rendering and the metadata labeled by VLM.

2) *Sample Prompt for ANYTASK*: Here are some examples of ANYTASK code generation prompt. Task generation prompts are shown in Figure 10. Success checker prompts are shown in Figure 11 and Figure 12 . ViPR policy prompts are shown in Figure 13 abd Figure 14. Observation state generation prompts are in Figure 16. Reward function prompts are in Figure 17 and Figure 18.

3) *Environment APIs*: Environment APIs are in Table XI. Detailed argument definitions are available in our code.

C. Trajectory Generation

1) *Skill APIs*: Please find skill APIs in Table XII. Detailed argument definitions are available in our code.

2) *ANYTASK Agents examples*: In this section, we present several examples generated by our system using ANYTASK agents. A ViPR example is illustrated in Figure 19, while a ViPR-RL example is provided in Figure 20.

3) *State, Reward, Success, and Domain Randomization*: These four Python functions—`compose_state`, `reward_function`, `check_success`, and `domain_randomize`—define the environment’s state representation, the task objective, the success condition, and

the environment initialization for reinforcement learning. These are important components for ViPR-EUREKA. A detailed example is in Figure 21.

4) *Object Manipulation Order Configuration*: This configuration defines the order in which objects must be manipulated to successfully complete the task. An example is shown in Figure 22.

D. Policy Learning

We train single task policies on our generated data and evaluate them in simulation. For simulation evaluation, we train a single-task 3D diffusion policy [64] on 500 demonstrations. The policy is conditioned on both a point cloud observation and on the robot’s current end-effector position and gripper state. We train each policy for 75,000 steps on one H100 GPU with a batch size of 1024, which takes approximately 8 hours. We use a learning rate of 0.000005 with a cosine schedule and the Adam optimizer [66]. We evaluate only the final checkpoint for each model. Our main results required training approximately 100 policies, requiring roughly 800 GPU hours.

E. Sim2Real

Figure 23 illustrates the point-cloud augmentation strategy used for sim-to-real transfer in the banana-on-can stacking task. The left panel shows the original point cloud rendered directly from simulation, which is clean and densely structured. The right panel shows the augmented point cloud that is actually fed into policy training. This augmented input includes global position and rotation jitter, simulated flying points to mimic sensor noise and outliers, and uniform downsampling to match real-world perception sparsity. As a reference, Figure 24 shows the real-world observations for drawer opening and stacking tasks. We observed that by better approximating real sensor artifacts, these augmentations improves robustness and generalization during real-world deployment.



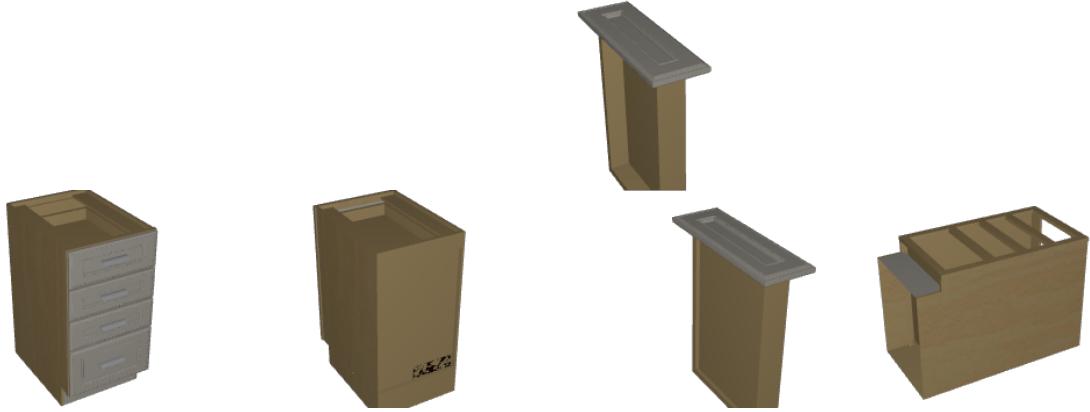
Fig. 7: A subset of the simulated assets (right) compared to real-world assets (left).

TABLE VIII: Task Table - Example Lifting Tasks

Task ID	Task Type	Description
3	ycb_lifting_1obj	Lift the tennis ball from the table surface by grasping it and holding it aloft without placing or stacking it.
7	ycb_lifting_1obj	Grasp the banana from the table and smoothly lift it upward to a height of approximately 10 cm above its initial position, holding it steady for a few seconds. This task focuses solely on the lifting motion without placing or stacking actions.
9	ycb_lifting_1obj	Grasp the apple and lift it vertically upward by approximately 0.15 m above its initial position.
10	ycb_lifting_1obj	Pick up and lift the philips screwdriver from the table.
16	ycb_lifting_1obj	Grasp the medium clamp from the table and lift it vertically upward until it is completely off the table surface, then hold it steady.
25	ycb_lifting_1obj	Grasp the flat screwdriver from the table and lift it upward without placing it back down.
27	ycb_lifting_1obj	Pick up the foam brick from its current position on the table and lift it vertically upward, holding it in the air without placing or stacking it on any other surface.
29	ycb_lifting_1obj	Lift the flat screwdriver from the table and hold it in the air at a fixed height.
14	ycb_lifting_2obj	Grasp the tennis ball from the table and lift it vertically upward, ensuring the ball is elevated away from the table surface.
15	ycb_lifting_2obj	Grasp the strawberry from the table and lift it gently upward, maintaining a secure grip during the lift.
0	ycb_lifting_3obj	Lift the large clamp by raising it vertically off the table by approximately 15 cm.
9	ycb_lifting_3obj	Grasp the orange and lift it vertically upward from its starting position on the table.
17	ycb_lifting_3obj	Lift the philips screwdriver by grasping it from the table and lifting it straight up to verify the grasp.

TABLE IX: Task Table - Example Pick and Place Tasks

Task ID	Task Type	Description
0	ycb_pick_and_place_1obj	Pick up the phillips screwdriver from its current location on the table and place it at a target location (e.g., at $x = 0.40$ m, $y = -0.45$ m) within the workspace.
14	ycb_pick_and_place_1obj	Pick up the apple and place it at a target location on the table (e.g., at $x = 0.5$, $y = -0.4$), ensuring that the apple is well within the robot's reachable workspace.
16	ycb_pick_and_place_1obj	Pick up the apple and place it at a central location within the workspace (e.g., $x: 0.4$, $y: -0.45$) to ensure clearance from the table edges.
23	ycb_pick_and_place_1obj	Pick up the potted meat can and place it at the front center of the workspace on the table.
2	ycb_pick_and_place_2obj	Pick up the fork and place it to the left of the plum on the table.
18	ycb_pick_and_place_2obj	Pick up the marbles and place them directly in front of the padlock on the table.
25	ycb_pick_and_place_2obj	Pick up the golf ball and place it in front of the padlock on the table.
37	ycb_pick_and_place_2obj	Pick up the power drill and place it to the right of the bleach cleanser on the table.
0	ycb_pick_and_place_3obj	Pick up the tennis ball and place it next to the pear.



Object Metadata:

```
{
  'name': 'storage_furniture',
  '_name_emb': <name language embedding omitted>,
  'desc': 'The object is a storage furniture piece featuring a rectangular design made from wood. It includes a total of four parts: one body and three drawers, each equipped with handles. The overall color is a light wood finish with neutral drawer fronts, providing a clean and functional appearance.',
  '_desc_emb': <desc language embedding omitted>,
  'shape': 'Rectangular, vertical storage unit with multiple drawers',
  '_shape_emb': <shape language embedding omitted>,
  'material': 'Wood',
  '_material_emb': <material language embedding omitted>,
  'bbox_desc': '1.1m long, 0.8m wide, and 1.5m tall.',
  '_bbox_desc_emb': <bbox language embedding omitted>,
  'pattern': 'Smooth surface with no distinct patterns',
  '_pattern_emb': <pattern language embedding omitted>,
  '_type': 'object',
  'bbox': (
    (-0.5500339754288982, -0.385874013080157, -0.8268710066308561),
    (0.5771890770736514, 0.4125760342546297, 0.7034910114080105)),
  'color': 'Light wood finish for the body; neutral color for the drawer fronts'
}
```

Fig. 8: Object Database Sample: Multi-view, Multi-part Rendering and VLM-labeled metadata

Object Metadata Continued (drawer part of the entire object):

```
{
  'label': 'drawer',
  'desc': 'The drawer is a rectangular component of the storage furniture, made from light brown wood. It features a solid color with a smooth surface. This drawer is designed to slide in and out, providing convenient storage space.',
  'shape': 'rectangular',
  'material': 'wood',
  'bbox_desc': '0.8m long, 0.3m wide, and 0.9m tall.',
  'pattern': 'solid',
  'name': 'link_1',
  'joints': ['joint_1', 'joint_6'],
  'bbox':
    ((-0.4042590431785981, -0.16426906793320312, -0.3997560500405213),
     (0.37755600785708515, 0.09633603205617348, 0.5206800517408057)),
  'color': 'light brown',
  'children': {
    'drawer_front_29':
      'desc': 'drawer front',
      'material': 'paint',
      'bbox_desc': '0.0cm long, 0.4m wide, and 5.6cm tall.',
      'color_desc': 'light beige',
      'pattern': 'solid',
      'texture': 'This material is a light beige paint with a solid pattern.',
      'bbox': ((-0.5028430304480387, -0.20997298721734534, -0.061841028196256004),
                (-0.5028430104844965, 0.23667601504659727, -0.006091010623107836)),
      'color': '#fcf1e4',
      '<... more children parts>
  }
}
```

Fig. 9: Object Database Sample (Continued)

TABLE X: Task Table - Example Pushing Tasks

Task ID	Task Type	Description
5	ycb_pushing_1obj	Push the tuna fish can using a steady forward push to a designated target location near the center of the workspace, ensuring controlled contact during motion.
10	ycb_pushing_1obj	Push the nine hole peg test object from its initial position to the center of the workspace (approximately x=0.4, y=-0.45) to simulate a controlled translational motion across the table.
17	ycb_pushing_1obj	Push the banana from its initial position to a target location near the center of the workspace (e.g., around x=0.4, y=-0.45) using a gentle linear pushing motion.
19	ycb_pushing_1obj	Push the mustard bottle across the table surface from its current position near the left corner towards the center of the workspace, ensuring controlled contact and avoiding excessive force that might tip the bottle.
20	ycb_pushing_1obj	Push the flat screwdriver along the table: starting from its initial position within the workspace, push it in a straight line along the positive x direction (forward) to a target location near the forward edge of the workspace. This task requires precise control to ensure the tool remains aligned while being pushed.
21	ycb_pushing_1obj	Push the pear gently along the positive x-axis (forward direction) until it reaches the center of the workspace.
24	ycb_pushing_1obj	Push the skillet lid towards the center of the workspace at approximately (0.4, -0.44), ensuring the movement is smooth and along the table surface.
39	ycb_pushing_2obj	Push the spoon such that it is repositioned to the left of the sponge (i.e., with a higher y value, closer to y = -0.2) to create a clear spatial separation between the two objects.
18	ycb_pushing_3obj	Push the pear along the x-axis so that it reaches a target zone near (x=0.50, y=-0.45) on the table.
22	ycb_pushing_3obj	Push the bowl from its current position to a target area, ensuring the marbles remain close by.

Task Generation:

You are a robotics research scientist responsible for developing manipulation skills for a robot. Given the following objects in the scene in a table-top environment, please propose some ycb pick and place 3obj tasks for training robot policies. You should also follow the format requirements:

1. You are expected to generate a ".json" format answer to describe each task. No other words outside .json part.
2. You should consider the capability of the robot.
3. Assume the positive z-axis is the up direction, positive x-axis is the forward direction, and the negative y-axis is the right direction, from the base of the robot.
4. Assume the robot is placed at (0, 0, 0).
5. Object ids should follow the numbers in the list of objects (which starts from 1).
6. Label the id of relevant objects (which will be manipulated, or will be considered in environment state) in the generated configuration.

Example:

Example 0

List of objects:

1. Apple
2. Banana
3. Cup

Environment:

Table-top manipulation environment. The table is placed in front of the robot.

Robot:

Degrees of Freedom: 7 DoF

Payload: Up to 3 kg

Reach: 855 mm

Precision: ±0.1 mm

Proposed 4 tasks:

```
{  
    "tasks": [  
        {  
            "task_description": "Pick up the apple and place it next to the banana.",  
            "relevant_object_ids": [1, 2]  
        },  
        {  
            "task_description": "Pick up the banana and place it in between the apple and the cup.",  
            "relevant_object_ids": [1, 2, 3]  
        },  
        {  
            "task_description": "Move the apple and place it behind the banana.",  
            "relevant_object_ids": [1, 2]  
        },  
        {  
            "task_description": "Place the apple to the right of the banana.",  
            "relevant_object_ids": [1, 2]  
        }  
    ]  
}
```

Fig. 10: Task Generation Prompt

Success Checker:

You are a robotics research scientist responsible for developing manipulation skills for a robot. Given the following objects and the robot manipulation task, please write a Python method called "check_success()" to indicate if an agent finishes the task successfully, and rephrase the task description that exactly matches your check success function, especially for some numerical values specified in the function but not in the original description.

You should give your response in json format "check_success": <your code>, "rephrased_task_description": <>

You should also follow the format requirements:

1. You are expected to generate Python code for "check_success".
2. You should consider the capability of the robot.
3. Assume the positive z-axis is the up direction, positive x-axis is the forward direction, and the negative y-axis is the right direction, from the base of the robot.
4. Assume the robot is placed at (0, 0, 0).
5. Please keep the object id consistent with the provided list of objects.
6. Write the function that is compatible with vectorized environments, i.e. variables are batched arrays (tensors). This is important for bool operations.

You are provided with following APIs to use

```
{  
    "type": "function",  
    "function": {  
        "name": "is_grasping_object",  
        "description": "Check if the robot is gripping the queried object (part).  
        Args:  
            env (Any): simulation environment  
            object_id (int): object id  
            part_name (str): name of the object part. Defaults to "" -- no object part specified.  
                If the object is an articulated object and the `part_name` is empty or not found,  
                return the gathered results of its parts by `or`.  
                If the object is a rigid object, ignore the `part_name`.  
            env_ids (torch.Tensor | None): if specified, return the information of env_ids only.  
        Returns:  
            torch.Tensor: True if the robot is gripping the object (part)"  
    }  
}
```

<More API descriptions omitted>

In addition, you are allow to use common functions in pytorch. Don't use other functions.

This is an example:

Environment:

Table-top manipulation environment. You can assume there will be a table. The closer right corner of the table is aligned with the robot. The table is 0.8m wide and 0.8m long.

Robot:

Degrees of Freedom: 7 DoF

Payload: Up to 3 kg

Reach: 855 mm

Precision: ±0.1 mm

List of objects:

1. Apple

2. Banana

3. Cup

Task description:

Pick up an apple.

Please generate the success check function:

Fig. 11: Success Checker Prompt

Success Checker Continued:

"check_success":

```
# return a bool tensor
def check_success(env) -> torch.Tensor:
    apple_id = 1
    robot_reach = 0.8 # approximation of the robot reach
    success_grasping = is_grasping_object(env, apple_id)
    # positions are represented by batched (x, y, z) coordinates
    # the height is z-axis
    apple_height = get_object_position(env, apple_id)[:, 2]
    apple_initial_height = get_object_init_position(env, apple_id)[:, 2]
    success_lifting = (apple_height - apple_initial_height) > (robot_reach * 0.5)
    # lift apple up of half of the robot reach
    success_lifting = success_lifting & (apple_height > 0) # apple is above the robot base
    return success_grasping & success_lifting
```

,

"rephrased_task_description": "pick up an apple and lift it up of half of the robot reach"

=====

Now here is the information for you:

Environment:

Table-top manipulation environment. You can assume there will be a table. The closer left corner of the table surface is aligned with the robot base (0, 0, 0). Assume the positive z-axis is the up direction, positive x-axis is the forward direction, and the negative y-axis is the right direction, from the base of the robot. Your work space is [0.15, 0.65] in x, [-0.68, -0.2] in y, in meters. This is for reference and you don't need to add a table.

Robot:

Degrees of Freedom: 7
Payload: Up to 3 kg
Reach: 855 mm
Precision: ± 0.1 mm

List of objects:

1. { 'object_name': 'skillet lid', 'object_description': 'skillet lid' }
2. { 'object_name': 'marbles', 'object_description': 'marbles' }
3. { 'object_name': 'cups', 'object_description': 'cups' }

Task description:

Pick up the marbles and place them next to the skillet lid.

Please generate the success check function:

Success Checker Response:

{ "check_success": "

...

...

}

Fig. 12: Success Checker Prompt Continued

VIPR Policy Prompt:

You are a robotics research scientist responsible for developing manipulation skills for a robot. Given the following objects, the robot manipulation task, and the corresponding success checker code, please write a Python method called "scripted_policy()" that executes a scripted policy to finish the task. The scripted policy is composed by a sequence of pre-defined skills with associated parameters. In order to better store the steps you solve the problem by scripted policy, use 'log_step_description' API to note down your explanation of the sub step in natural language sentences. Refer to this API's docstring for the usage guide. You are encouraged to give the exact values when necessary and the information is easy to get from environment APIs. Python's f-string is useful. Based on the scripted policy you generated, please also label the ordering of the objects being manipulated. Your final output format of this request is **json format** like

```
{"scripted_policy": ..., "object_manipulation_order": [{"object_id": <object id, should be consistent with the information in given list of objects>, "part_name": <if the object is articulated object, specify the part it wants to manipulate. Use null if there is no>, "joint_name": <if the object is articulated object, specify the joint that related to manipulating that part, if the joint is not a fixed joint. Use null if there is not>}]}
```

where 'scripted_policy' contains your code and 'object_manipulation_order' contains the object manipulation order. You should also follow the format requirements:

1. You are expected to generate a Python format answer. No other words outside Python part.
2. You should consider the capability of the robot.
3. Assume the positive z-axis is the up direction, positive x-axis is the forward direction, and the negative y-axis is the right direction, from the base of the robot.
4. Assume the robot is placed at (0, 0, 0).
5. Please keep the object id consistent with the provided list of objects.
6. Write the function that is compatible with vectorized environments, i.e. variables are batched arrays (tensors). This is important for bool operations.
7. Try to match your scripted policy with the success checker so that your policy can success.
8. No pre-grasp steps.
9. For tasks involving operating articulated object, it's good to use the API to check the current joint position and joint limits, then decide the parameters for skills, but you don't need to use a loop. You can determine the value and execute it by one shot. Avoid guessing how much you should operate. Also, don't use the axis provided in joint attribute to determine the operation direction of the joint because the orientation of the object changes. Use API instead.

You are provided with following skill APIs:

```
{  
    "type": "function",  
    "function": {  
        "name": "move_to",  
        "description": "Move the robot end effector to target position and orientation, with the gripper status unchanged.  
        Args:  
            env (Any): simulation environment  
            target_position (torch.Tensor): target position to move, shape (N, 3)  
            target_orientation (torch.Tesnor | None): target orientation in quat, shape (N, 4), or None  
                Defaults to None (keep the current orientation).  
            gripper_open (bool): `True` to open the gripper, `False` to close the gripper.  
                Default to True.",  
        "parameters": {"type": "object", "properties": {  
            "env": {"type": "Any"},  
            "target_position": {"type": "torch.Tensor"},  
            "target_orientation": {"type": "torch.Tensor | None"},  
            "gripper_open": {"type": "bool"}},  
            "required": ["env", "target_position", "target_orientation"]}}}
```

<More skill APIs omitted>

Fig. 13: VIPR Policy Prompt

ViPR Policy Prompt Continued:

In addition, you are allowed to use common functions in pytorch.

This is an example:

Environment: Table-top manipulation environment. You can assume there will be a table. The closer right corner of the table is aligned with the robot. The table is 0.8m wide and 0.8m long.

Robot: Degrees of Freedom: 7 DoF Payload: Up to 3 kg Reach: 855 mm Precision: ± 0.1 mm

List of objects:

1. Apple
2. Banana
3. Cup

Task description:

Place the apple next to the banana.

<Success checker code omitted>

Please generate the scripted policy:

```
{"scripted_policy": "
    import torch
    ...
    ...
    object_manipulation_order": [{"object_id": 1, "part_name": null, "joint_name": null}
}
```

=====

Now here is the information for you: Environment:

Table-top manipulation environment. You can assume there will be a table. The closer left corner of the table surface is aligned with the robot base (0, 0, 0). Assume the positive z-axis is the up direction, positive x-axis is the forward direction, and the negative y-axis is the right direction, from the base of the robot. Your work space is [0.15, 0.65] in x, [-0.68, -0.2] in y, in meters. This is for reference and you don't need to add a table.

Robot:

Degrees of Freedom: 7 DoF

Payload: Up to 3 kg

Reach: 855 mm

Precision: ± 0.1 mm

List of objects:

1. 'object_name': 'skillet lid', 'object_description': 'skillet lid'
2. 'object_name': 'marbles', 'object_description': 'marbles'
3. 'object_name': 'cups', 'object_description': 'cups'

Task description:

Pick up the skillet lid and place it on top of the cups. The task is considered successfully completed if the robot is no longer grasping the skillet lid, the lid is positioned within 0.05 meters horizontally (in the x-y plane) of the cups, and its height is at least 0.02 meters above that of the cups.

Please generate the scripted policy:

ViPR Policy Response:

```
import torch
...
...
"object_manipulation_order": [{"object_id": 1, "part_name": null, "joint_name": null}]
```

Fig. 14: ViPR prompt continued

Compose State Prompt:

You are a robotics research scientist responsible for developing manipulation skills for a robot. Given the following objects, the robot manipulation task, and the corresponding success checker code, please write a Python function `compose_state()` to compose state information required to successfully train an RL policy. You should also follow the format requirements:

1. You are expected to generate a Python format answer. No other words outside Python part.
2. You should consider the capability of the robot.
3. Assume the positive z-axis is the up direction, positive x-axis is the forward direction, and the negative y-axis is the right direction, from the base of the robot.
4. Assume the robot is placed at (0, 0, 0).
5. Object ids should follow the numbers in the list of objects (which starts from 1).
6. We already provide the state information of robot itself. You will be responsible for object states or robot-object states.
7. At least cover all relevant objects.
8. You can compose any privileged state information that is helpful for RL, as long as only the provided APIs and common operations are used.
9. Write the function that is compatible with vectorized environments, i.e. variables are batched arrays (tensors). This is important for bool operations.
10. The definition of the `compose_state()` is ‘`def compose_state(env) -> dict[str, torch.Tensor]:`‘, where ‘`env`‘ is the simulation environment and the return value is a dictionary where the key is the name of the state and the value is the state information (in `torch.Tensor`). Don’t use duplicated keys.
11. You are not able to access `check_success()` directly.

You are provided with following APIs to use

<APIs are omitted>

=====

In addition, you are allow to use common functions in pytorch.

Example:

List of objects:

1. Apple 2. Banana 3. Cup

Environment:

Table-top manipulation environment. The table is placed in front of the robot.

Robot: Degrees of Freedom: 7 DoF Payload: Up to 3 kg Reach: 855 mm Precision: ±0.1 mm

Task description:

Pick up the apple and place it next to the banana.

Relevant object ids: [1, 2]

Please generate the ‘`compose_state`‘ function:

```
def compose_state(env)-> dict[str, torch.Tensor]:  
    # relevant objects  
    apple_id = 1  
    banana_id = 2  
    # end-effector to apple distance  
    eef_to_apple = object_to_robot_eef_distance(env, apple_id)  
    # end-effector to banana distance  
    eef_to_banana = object_to_robot_eef_distance(env, banana_id)  
    # apple to banana  
    apple_to_banana = distance(get_object_position(env, apple_id),  
        get_object_position(env, banana_id))  
    return {  
        "eef_to_apple": eef_to_apple,  
        "eef_to_banana": eef_to_banana,  
        "apple_to_banana": apple_to_banana  
    }
```

=====

Fig. 15: Compose state prompt

Compose State Continued:

Now here is the task:

List of objects:

1. {'object_name': 'skillet lid', 'object_description': 'skillet lid'}
2. {'object_name': 'marbles', 'object_description': 'marbles'}
3. {'object_name': 'cups', 'object_description': 'cups'}

Environment:

Table-top manipulation environment. You can assume there will be a table. The closer left corner of the table surface is aligned with the robot base (0, 0, 0). Assume the positive z-axis is the up direction, positive x-axis is the forward direction, and the negative y-axis is the right direction, from the base of the robot. Your work space is [0.15, 0.65] in x, [-0.68, -0.2] in y, in meters. This is for reference and you don't need to add a table.

Robot:

Degrees of Freedom: 7 DoF

Payload: Up to 3 kg

Reach: 855 mm

Precision: ± 0.1 mm

Task description:

Pick up the marbles and place them next to the skillet lid such that the marbles, once released by the robot, are within 0.15 meters of the skillet lid in Euclidean distance and have a vertical (z-axis) difference smaller than 0.1 meters.

Relevant object ids: [1, 2]

Success checker code:

Compose State Response:

```
import torch
def check_success(env) -> torch.Tensor:
    # Object ids
    skillet_lid_id = 1
    marbles_id = 2
    # Thresholds
    placement_threshold = 0.15
    # max horizontal distance for marbles to be considered next to skillet lid (in meters)
    vertical_threshold = 0.1    # max allowed difference in z-axis (in meters)
    # Get current positions of the objects; shape: (N, 3)
    marbles_pos = get_object_position(env, marbles_id)
    skillet_lid_pos = get_object_position(env, skillet_lid_id)
    # Compute Euclidean distance between marbles and skillet lid
    dist = distance(marbles_pos, skillet_lid_pos)
    # Check if horizontal placement is within threshold
    placement_success = dist < placement_threshold
    # Check vertical alignment: the z difference should be small (within vertical_threshold)
    z_diff = torch.abs(marbles_pos[:, 2] - skillet_lid_pos[:, 2])
    vertical_alignment = z_diff < vertical_threshold
    # Ensure that the marbles are not currently being grasped
    not_grasping = ~is_grasping_object(env, marbles_id, "")
    # Overall success: marbles placed next to the skillet lid
    # (both horizontal and vertical conditions) and released
    success = placement_success & vertical_alignment & not_grasping
    return success
```

Fig. 16: Compose state prompt continued

Reward Function:

You are a robotics research scientist responsible for developing manipulation skills for a robot. Given the following objects, the robot manipulation task, and the corresponding success checker code, please write a Python method called "reward_function()" to assign rewards to an RL agent that is learning this manipulation task. You should also follow the format requirements:

1. You are expected to generate a Python format answer. No other words outside Python part.
2. You should consider the capability of the robot.
3. Assume the positive z-axis is the up direction, positive x-axis is the forward direction, and the negative y-axis is the right direction, from the base of the robot.
4. Assume the robot is placed at (0, 0, 0).
5. Please keep the object id consistent with the provided list of objects.
6. Write the function that is compatible with vectorized environments, i.e. variables are batched arrays (tensors). This is important for bool operations.
7. Try your best to design the reward function with the reference of success condition (success checker). But you are not able to access check_success() directly.

You are provided with following APIs to use:

<Environment APIs are omitted>

In addition, you are allowed to use common functions in pytorch. This is an example:

Environment:

Table-top manipulation environment. You can assume there will be a table. The closer right corner of the table is aligned with the robot. The table is 0.8m wide and 0.8m long.

Robot:

Degrees of Freedom: 7 DoF

Payload: Up to 3 kg

Reach: 855 mm

Precision: ±0.1 mm

List of objects:

1. Apple

2. Banana

3. Cup

Task description:

Pick up an apple.

Please generate the reward function:

```
def reward_function(env):  
    objects = get_object_list(env)  
    # apple's object id is 1  
    apple_id = 1  
    # query the distance to robot end effector  
    reward = zero_rewards(env)  
    env_is_grasping_object = is_grasping_object(env, apple_id)  
    # success grasping  
    reward[env_is_grasping_object] += 100  
    # apple is being picked up  
    current_apple_position = get_object_position(env, apple_id)  
    init_apple_position = get_object_initial_position(env, apple_id)  
    total_delta_z = current_apple_position[:, 2] - init_apple_position[:, 2]  
    reward[env_is_grasping_object & (total_delta_z > 0.3)] += 10000  
    return reward
```

=====

Fig. 17: Reward function prompt.

Reward Function Continued:

Now here is the information for you:

Environment:

Table-top manipulation environment. You can assume there will be a table. The closer left corner of the table surface is aligned with the robot base (0, 0, 0). Assume the positive z-axis is the up direction, positive x-axis is the forward direction, and the negative y-axis is the right direction, from the base of the robot. Your work space is [0.15, 0.65] in x, [-0.68, -0.2] in y, in meters. This is for reference and you don't need to add a table.

Robot:

Degrees of Freedom: 7 DoF

Payload: Up to 3 kg

Reach: 855 mm

Precision: ± 0.1 mm

List of objects:

1. {'object_name': 'skillet lid', 'object_description': 'skillet lid'}

2. {'object_name': 'marbles', 'object_description': 'marbles'}

3. {'object_name': 'cups', 'object_description': 'cups'}

Task description:

Pick up the skillet lid and place it on top of the cups. The task is considered successfully completed if the robot is no longer grasping the skillet lid, the lid is positioned within 0.05 meters horizontally (in the x-y plane) of the cups, and its height is at least 0.02 meters above that of the cups.

<Success checker code omitted>

Please generate the reward function:

Reward function response:

```
def reward_function(env):
    import torch
    ...
    ...
    return reward
```

Fig. 18: Reward function prompt continued.

TABLE XI: List Environment APIs

API	Description
get_object_position()	Object position
get_object_initial_position()	Object position at the beginning of the episode
get_object_last_position()	Object position at last simulation step
get_object_rotation()	Object orientation
get_object_initial_rotation()	Object orientation at the beginning of the episode
get_object_last_rotation()	Object orientation at last simulation step
get_contact_grasp_position_external()	Grasp pose wrt an object
check_no_collision()	Check if the objects are colliding
get_workspace_limit()	Obtain the workspace limits
sample_position_in_workspace()	Sample a position within the workspace (for placing the object)
check_within_workspace()	Check if the object is within the workspace
add_object_to_env()	Place an object into the scene
get_object_list()	Get the list of objects
point_to_object_distance()	Distance between the point and an object
get_robot_joint_positions()	Robot joint positions
get_robot_eef_orientation()	Robot eef orientation
get_robot_eef_position()	Robot eef position
get_robot_eef_last_position()	Robot eef position at last simulation step
get_robot_eef_initial_position()	Robot eef position at the beginning of the episode
distance()	Distance between two objects
object_to_robot_eef_distance()	Distance between the object and the robot eef
point_to_robot_eef_distance()	Distance between a point and the robot eef
is_grasping_object()	Is the robot grasping the object
success_grasp()	Has the robot successfully grasped the object
maintain_grasp()	Is the robot maintaining the grasping
drop_grasp()	Has the robot dropped the object
get_object_bbox()	Object bounding box
get_articulated_obj_joint_state()	State of a joint in an articulated object
get_articulated_obj_joint_limits()	Limits of a joint in an articulated object
get_articulated_obj_joint_operating_direction()	Operating direction of a joint in an articulated object
get_articulated_obj_part_pos()	Position of a part in an articulated object
get_articulated_obj_part_rotation()	Orientation of a part in an articulated object
get_articulated_obj_part_bbox()	Bounding box of a part in an articulated object
sample_orientation_facing_towards_robot()	Sample an orientation that makes the object face towards the robot
zero_rewards()	Create a zero reward tensor
log_step_description()	Log the language description of a scripted policy step
make_a_tensor()	Make a tensor

TABLE XII: List of Skill APIs

API	Description
reset_robot()	Move the robot to home position
pick()	Pick up an object
close_gripper()	Close gripper
open_gripper()	Open gripper
place()	Place object
push_or_pull_object_to_xy()	Push or pull the object
move_to()	Move the robot to a position

ViPR

```

import torch

def scripted_policy(env):
    # Object IDs
    softball_id = 1
    clamp_id = 2
    # Get initial positions for the softball and the robot's end effector
    softball_position = get_object_position(env, softball_id) # shape (N,3)
    robot_eef_position = get_robot_eef_position(env)
    # Log the state before picking up the softball
    log_step_description(
        env,
        before_step={
            "template": "Softball is at {softball_position} and robot EEF is at {robot_eef_position} before grasping.",
            "kwargs": {
                "softball_position": softball_position,
                "robot_eef_position": robot_eef_position
            }
        },
        action_todo={
            "template": "Approach the softball to grasp it.",
            "kwargs": {}
        }
    )
    # Approach Phase: move above the softball
    hover_offset = torch.tensor([[0.0, 0.0, 0.1]]), device=softball_position.device) # 10 cm above
    above_softball = softball_position + hover_offset
    move_to(env, target_position=above_softball, target_orientation=None, gripper_open=True)
    # Move down to the softball to grasp it
    move_to(env, target_position=softball_position, target_orientation=None, gripper_open=True)
    # Grasp the softball by closing the gripper
    close_gripper(env)
    # Lift Phase: lift the softball by moving back to the above position
    move_to(env, target_position=above_softball, target_orientation=None, gripper_open=False)
    # Determine target placement: place softball on top of extra large clamp
    clamp_position = get_object_position(env, clamp_id) # shape (N,3)
    # Set target such that the horizontal (x, y) centers align with clamp, and elevate by additional 0.05 m
    target_position = torch.cat([clamp_position[:, :2], clamp_position[:, 2:3] + 0.05], dim=1)
    # Log the state before placing the softball
    log_step_description(
        env,
        before_step={
            "template": "Preparing to place softball: clamp is at {clamp_position}, target placement is {target_position}.",
            "kwargs": {
                "clamp_position": clamp_position,
                "target_position": target_position
            }
        },
        action_todo={
            "template": "Move to target placement and release the softball.",
            "kwargs": {}
        }
    )
    # Move above the target placement
    above_target = target_position + hover_offset
    move_to(env, target_position=above_target, target_orientation=None, gripper_open=False)
    # Lower the softball to the target placement on top of the clamp
    move_to(env, target_position=target_position, target_orientation=None, gripper_open=False)
    # Release Phase: open the gripper to drop the softball
    open_gripper(env)
    # Retract: move back up to above the target placement
    move_to(env, target_position=above_target, target_orientation=None, gripper_open=True)
    # Log the final state after placing the softball
    log_step_description(
        env,
        after_step={
            "template": "After placing, softball is at {softball_pos} and clamp is at {clamp_pos}.",
            "kwargs": {
                "softball_pos": get_object_position(env, softball_id),
                "clamp_pos": clamp_position
            }
        }
    )
)

```

Fig. 19: Generated policy demonstrating a sequence of robotic actions to pick up a softball and place it on a clamp.

ViPR-RL

```

import torch

def scripted_policy_rl(env):
    # Object IDs
    softball_id = 1
    clamp_id = 2

    # ----- Part 1: Pick up the softball -----
    # Use contact sampling to get the grasp pose for the softball via RL-based skill
    grasp_position, grasp_orientation = get_contact_grasp_position_external(env, softball_id, part_name="")

    # Log before picking
    log_step_description(
        env,
        before_step={
            "template": "Softball grasp pose computed: position {grasp_position}, orientation {grasp_orientation}.",
            "kwargs": {"grasp_position": grasp_position, "grasp_orientation": grasp_orientation}
        },
        action_todo={
            "template": "Moving to grasp pose for softball.",
            "kwargs": {}
        }
    )

    # Move the robot's end effector to the computed grasp pose
    move_to(
        env=env,
        target_position=grasp_position,
        target_orientation=grasp_orientation,
        gripper_open=True
    )

    # Execute the RL-based picking skill for the softball
    pick_success = pick_rl(env, external_id=softball_id)

    # Log after pick
    log_step_description(
        env,
        after_step={
            "template": "RL pick executed for softball (ID {softball_id}). Success: {pick_success}.",
            "kwargs": {"softball_id": softball_id, "pick_success": pick_success}
        }
    )

    # ----- Part 2: Place the softball on the extra large clamp -----
    # Get the current position of the clamp
    clamp_position = get_object_position(env, clamp_id)

    # Define placement: center the softball on the clamp with a vertical offset to satisfy stacking constraints
    # Horizontal alignment: within 0.05m is required. We choose the clamp center.
    # Vertical offset: at least 0.03m above the clamp. We use 0.05m as a safe offset.
    placement_vertical_offset = 0.05
    target_position = clamp_position + torch.tensor([[0.0, 0.0, placement_vertical_offset]]), device=clamp_position.device

    # Define an above-placement position, by adding an extra vertical offset (e.g., 0.1 m)
    above_offset = torch.tensor([[0.0, 0.0, 0.1]]), device=target_position.device
    above_target_position = target_position + above_offset

    # Log before placing
    log_step_description(
        env,
        before_step={
            "template": "Target placement for softball on clamp: position {target_position}.",
            "kwargs": {"target_position": target_position}
        },
        action_todo={
            "template": "Moving to placement pose above the clamp.",
            "kwargs": {}
        }
    )

    # Move above the placement position (with the object in hand)
    move_to(
        env=env,
        target_position=above_target_position,
        target_orientation=None,
        gripper_open=False
    )

    # Move down to the target placement position
    move_to(
        env=env,
        target_position=target_position,
        target_orientation=None,
        gripper_open=False
    )

    # Open the gripper to release the softball
    open_gripper(env)

    # Lift the robot's end effector upward after releasing
    move_to(
        env=env,
        target_position=above_target_position,
        target_orientation=None,
        gripper_open=True
    )

    # Log after placing
    log_step_description(
        env,
        after_step={
            "template": "Softball placed on clamp. Softball position: {softball_pos}, Clamp position: {clamp_pos}.",
            "kwargs": {"softball_pos": get_object_position(env, softball_id), "clamp_pos": clamp_position}
        }
    )

```

Fig. 20: Generated ViPR-RL policy for the robot softball placement task. `pick_rl` is the invoked RL skill.

ViPR-EUREKA

```

def compose_state(env) -> dict[str, torch.Tensor]:
    import torch
    # Relevant object IDs for the task
    softball_id = 1
    clamp_id = 2

    # Get the current positions of the softball and the clamp
    pos_softball = get_object_position(env, softball_id) # shape (N, 3)
    pos_clamp = get_object_position(env, clamp_id) # shape (N, 3)

    # Compute horizontal (x-y plane) difference between softball and clamp
    horizontal_diff = torch.norm(pos_softball[:, :2] - pos_clamp[:, :2], dim=1, keepdim=True)
    # Compute vertical difference (z dimension) between softball and clamp
    vertical_diff = (pos_softball[:, 2:3] - pos_clamp[:, 2:3])

    # Compute distance from robot end-effector to the softball and to the clamp
    eef_to_softball = object_to_robot_eef_distance(env, softball_id).unsqueeze(1) # shape (N,1)
    eef_to_clamp = object_to_robot_eef_distance(env, clamp_id).unsqueeze(1) # shape (N,1)

    # Compose state dictionary with object positions and key distances
    state = {
        "softball_position": pos_softball,
        "clamp_position": pos_clamp,
        "horizontal_diff": horizontal_diff,
        "vertical_diff": vertical_diff,
        "eef_to_softball": eef_to_softball,
        "eef_to_clamp": eef_to_clamp,
    }

    return state
#####
def reward_function(env):
    import torch
    # Object ids based on the provided list:
    # softball: 1, extra large clamp: 2
    softball_id = 1
    clamp_id = 2

    # Initialize reward tensor as zero
    reward = zero_rewards(env)

    # get the current positions of the softball and the clamp (shape: (N, 3))
    pos_softball = get_object_position(env, softball_id)
    pos_clamp = get_object_position(env, clamp_id)

    # compute the horizontal (x-y) distance between the centers of softball and clamp
    horizontal_distance = torch.norm(pos_softball[:, :2] - pos_clamp[:, :2], dim=1)

    # compute the vertical difference (z-axis): softball - clamp
    vertical_diff = pos_softball[:, 2] - pos_clamp[:, 2]

    # Shaping rewards:
    # Encourage the agent to minimize the difference to the target horizontal offset (0.05 m).
    r_horizontal = 1.0 - torch.tanh(10.0 * torch.abs(horizontal_distance - 0.05))
    # Encourage lifting the softball above the clamp.
    r_vertical = torch.tanh(5.0 * (vertical_diff - 0.03))

    shaping_reward = r_horizontal + r_vertical
    reward = reward + shaping_reward

    # Bonus reward: if the softball is well stacked on the clamp per the success conditions.
    success = (horizontal_distance < 0.05) & (vertical_diff > 0.03)
    reward[success] += 10000.0

    # Additional bonus if the robot is currently grasping the softball (encourage pickup)
    grasping = is_grasping_object(env, softball_id)
    reward[grasping] += 100.0

    return reward
#####
def check_success(env) -> torch.Tensor:
    import torch
    # Object ids for the task
    softball_id = 1
    clamp_id = 2

    # Get current positions of the softball and the extra large clamp
    pos_softball = get_object_position(env, softball_id) # shape (N, 3)
    pos_clamp = get_object_position(env, clamp_id) # shape (N, 3)

    # Compute the horizontal distance (in the x-y plane) between the softball and the clamp
    horizontal_diff = torch.norm(pos_softball[:, :2] - pos_clamp[:, :2], dim=1)

    # Compute the vertical difference (z-axis). The softball should be higher than the clamp
    vertical_diff = pos_softball[:, 2] - pos_clamp[:, 2]

    # Define thresholds: horizontal difference must be less than 0.05 m and vertical difference more than 0.03 m
    success_horizontal = horizontal_diff < 0.05
    success_vertical = vertical_diff > 0.03

    # The task is successful if both conditions are met
    return success_horizontal & success_vertical
#####
def domain_randomize(env, env_ids: torch.Tensor) -> torch.Tensor:
    import torch
    import numpy as np

    # Get workspace limits: workspace is defined for x and y.
    workspace_limit = get_workspace_limit(env)
    # Number of environments
    N = len(env_ids)
    # Initialize success flag for each environment as True
    success = torch.ones(N, device=env.device, dtype=torch.bool)

    # We'll assume the table-top height is fixed at 0.0 (i.e. objects are placed on the table surface).
    table_z = 0.0

    # Object ids:
    # 1: softball (target), 2: extra large clamp (support), 3: spoon (irrelevant)

    # Place the extra large clamp first.
    clamp_id = 2
    clamp_pos = sample_position_in_workspace(env, workspace_limit, env_ids=env_ids)
    clamp_pos[:, 2] = table_z
    success = success & add_object_to_env(env, clamp_id, clamp_pos, env_ids=env_ids)

    # Place the softball.
    softball_id = 1
    softball_pos = sample_position_in_workspace(env, workspace_limit, env_ids=env_ids)
    softball_pos[:, 2] = table_z
    success = success & add_object_to_env(env, softball_id, softball_pos, env_ids=env_ids)

    # Place the spoon (irrelevant object) at a random position.
    spoon_id = 3
    spoon_pos = sample_position_in_workspace(env, workspace_limit, env_ids=env_ids)
    spoon_pos[:, 2] = table_z
    success = success & add_object_to_env(env, spoon_id, spoon_pos, env_ids=env_ids)

    return success

```

Fig. 21: Basic components from ViPR-EUREKA , including the environment `compose_state`, the `reward_function`, the `check_success` function, and the `domain_randomize` functions.

Manipulation Order

```

"object_manipulation_order": [
    {
        "object_id": 1,
        "part_name": null,
        "joint_name": null
    },
    {
        "object_id": 2,
        "part_name": null,
        "joint_name": null
    }
],

```

Fig. 22: The object_manipulation_order configuration, specifying that object 1 (softball) must be handled before object 2 (clamp) in the manipulation , which will be used as an input to the contact sampling.

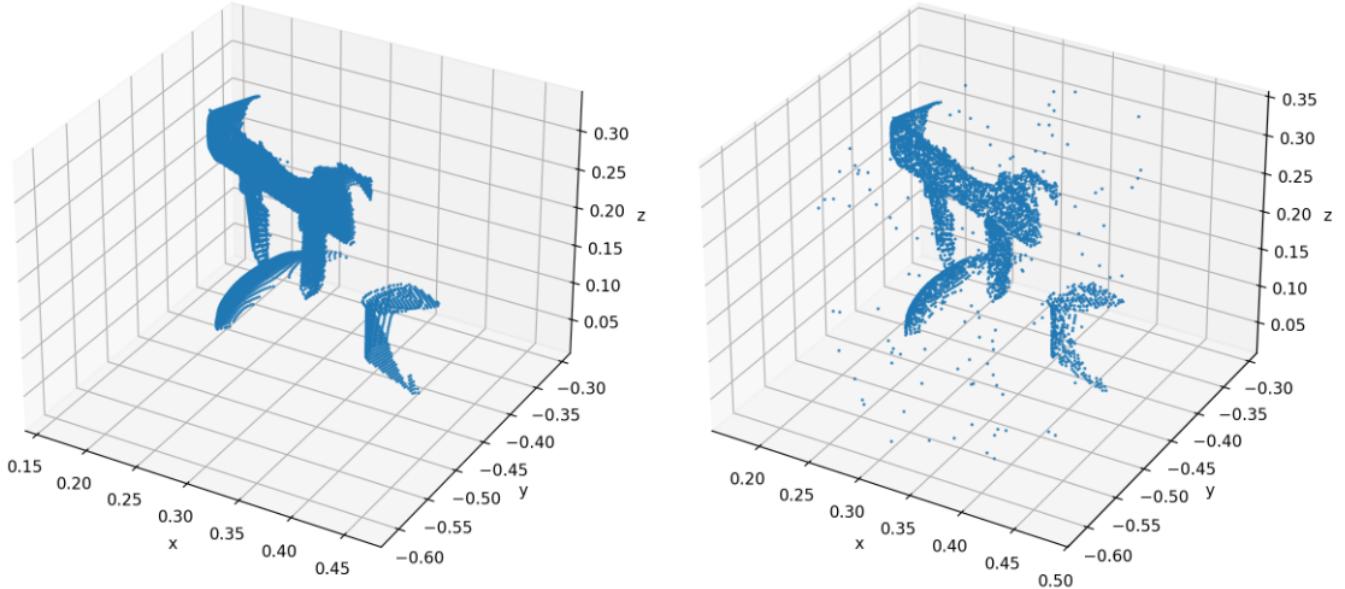


Fig. 23: Point-cloud augmentation for robust sim-to-real transfer.

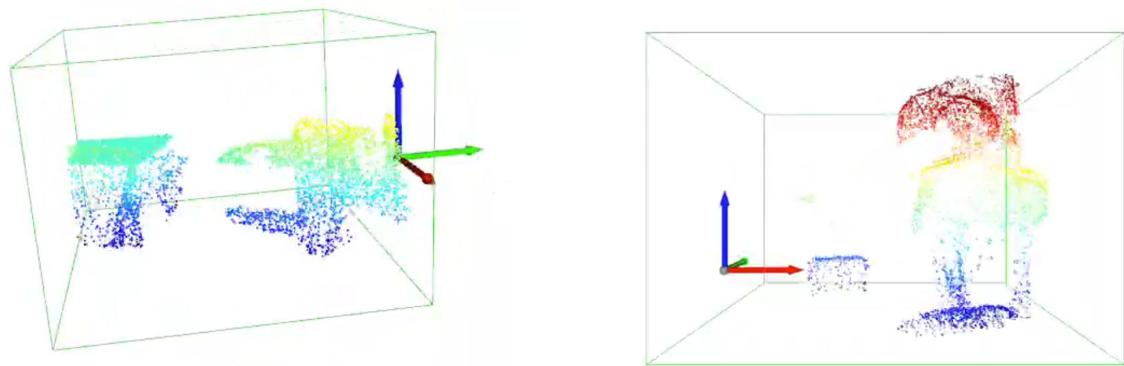


Fig. 24: Point-cloud observations during real-world deployment.

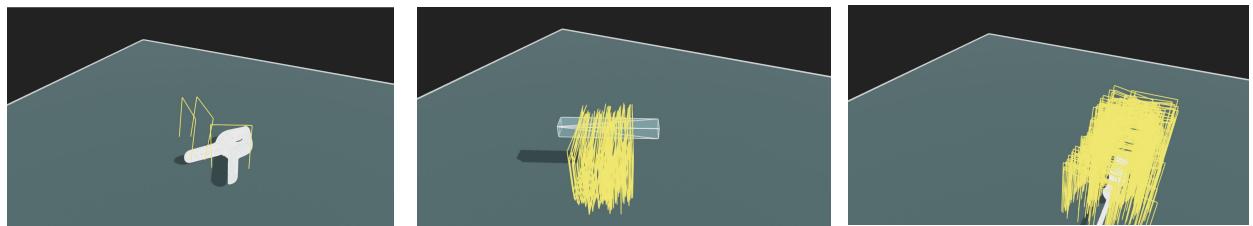


Fig. 25: Visualizations of contact sampling results for the clamp (left), drawer handle collision mesh (middle), and screwdriver (right).