Supplementary Material

Table of Contents

A Overview

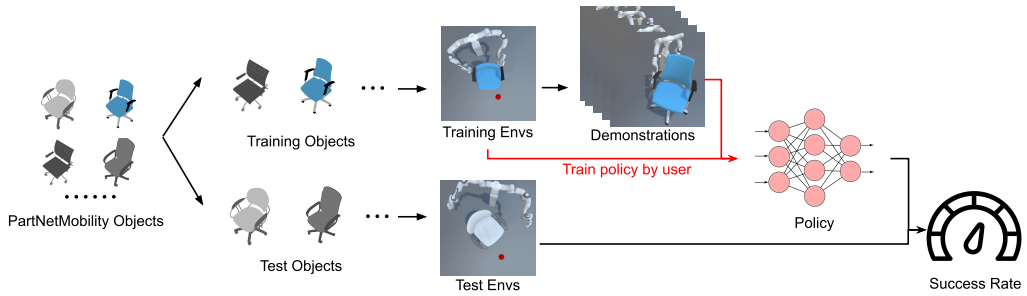

Figure 5: Overall illustration of ManiSkill. We manually re-model and postprocess objects from the PartNet-Mobility dataset, split them into training and test sets, and then build the corresponding training and test environments in the SAPIEN simulator. We then generate successful demonstration trajectories on the training environments. Users are expected to build policies based on the demonstration trajectories and the training environments, then evaluate the *mean success rate* on the test environments with the provided evaluation kit.

This supplementary material includes details on the system and task design of ManiSkill, along with implementations of baseline experiments. An overall architecture of our ManiSkill Benchmark presented in the main paper is summarized in Figure 5.

Section B and C provide more details on the system, tasks, and demonstration collection.

Section D.1 provides implementation details of point cloud subsampling in our baselines.

Section D.2 provides implementation details of our point cloud-based baseline network architectures, along with a diagram of our PointNet + Transformer model.

Section D.3 provides implementation details of learning-from-demonstrations algorithms, specifically imitation learning (Behavior Cloning) and offline RL.

B Further Details of Tasks and System

B.1 Environment Speed

Observation Mode	`state`	`pointcloud/rgbd`
OpenCabinetDoor	112 ± 4	48 ± 3
OpenCabinetDrawer	113 ± 4	47 ± 2
PushChair	53 ± 9	31 ± 4
MoveBucket	61 ± 7	34 ± 3

Table 4: Mean and standard deviation of FPS (frame per second) of the environments in ManiSkill. In `state` mode, most computations are used on physical simulation. In `pointcloud` and `rgbd` modes, most computations are used on rendering. All the numbers are tested on a single Intel i9-9960X CPU and a single NVIDIA RTX TITAN GPU.

We provide the running speeds of our environments in Table 4. We would like to note that the numbers cannot be directly compared with other environments like the MuJoCo tasks in OpenAI gym based on the following reasons.

1. In our environments, one environment step corresponds to 5 control steps. This "frame-skipping" technique can make the horizon of our tasks shorter, which is also a common practice in reinforcement learning [45]. We can make the environment FPS 5x larger by simply disabling frame-skipping, but this will make the tasks more difficult as the agent needs to make more decisions.

2. Compared to other environments, such as the MuJoCo tasks in OpenAI Gym, our articulated objects and robots are much more complicated. For example, our chairs contain up to 20 joints and tens of thousands of collision meshes. Therefore, the physical simulation process is inherently slow.
3. Many other robotics/control environments do not provide visual observations, while ManiSkill does. When generating visual observations, rendering is a very time-consuming process, especially when we are using **three** cameras simultaneously.

B.2 Environment Parameters

For each environment $\mathcal{E}_o$ and its associated object o, the environment parameters $\mathcal{L}_o$ provide randomization for our robot and for the target object.

For our robot, we randomly initialize the robot's in-plane position (x, y) and orientation (rotation around z-axis). Joints on the robot arms are initialized to a canonical pose, which is a common practice in robotics tasks.

For the target object, several physical parameters, such as joint frictions, are randomized for all tasks. In MoveBucket and PushChair, we also randomize the initial in-plane position (x, y) and the initial orientation (rotation around z-axis) of the bucket and the chair. Detailed implementations can be found in our repository `https://github.com/haosulab/ManiSkill`.

B.3 Segmentation Masks

As mentioned in Sec 2.3, we provide task-relevant segmentation masks in `pointcloud` and `rgbd` modes. Each mask is a binary array indicating a part or an object. Here are the details about our segmentation masks for each task:

- OpenCabinetDoor: handle of the target door, target door, robot (3 masks in total)
- OpenCabinetDrawer: handle of the target drawer, target drawer, robot (3 masks in total)
- PushChair: robot (1 mask in total)
- MoveBucket: robot (1 mask in total)

Basically, we provide the robot mask and any mask that is necessary for specifying the target. For example, in OpenCabinetDoor/Drawer environments, a cabinet might have many doors/drawers, so we provide the door/drawer mask such that users know which door/drawer to open. We also provide handle mask such that users know from which direction the door/drawer should be opened.

B.4 Success Conditions

OpenCabinetDoor and **OpenCabinetDrawer** Success is marked by opening the joint to 90% of its limit and keeping it static for a period of time afterwards. We do not constrain how the door/drawer is opened, e.g, the door can be opened by grabbing the side of the door or grasping the handle.

PushChair The task is successful if the chair (1) is close enough (within 15 centimeters) to the target location; (2) is kept static for a period of time after being close enough to the target; and (3) does not fall over.

MoveBucket The task is successful if (1) the bucket is placed on or above the platform at the upright position and kept static for a period of time, and (2) all the balls remain in the bucket.

In all tasks, the time limit for each episode is 200, which is sufficient to solve the task. An episode will be evaluated as unsuccessful if it goes beyond the time limit.

B.5 Evaluation Kit

ManiSkill provides a straightforward evaluation script. The script takes a task name, an observation mode (RGB-D or point cloud), and a solution file as input. The solution file is expected to contain a single policy function that takes observations as input and outputs an action. The evaluation kit takes

the policy function and evaluates it on the test environments. For each environment, it reports the average success rate and the average satisfactory rate for each success conditions (e.g. whether the ball is inside the bucket and whether the bucket is on or above the platform in MoveBucket).

B.6 System Design

We configure our simulation environments by a YAML-based configuration system. This system is mainly used to configure physical properties, rendering properties, and scene layouts that can be reused across tasks. It allows benchmark designers to specify simulation frequencies, physical solver parameters, lighting conditions, camera placement, randomized object/robot layouts, robot controller parameters, object surface materials, and other common properties shared across all environments. After preparing the configurations, designers can load the configurations as SAPIEN scenes and perform further specific customization with Python scripts. In our task design, after we build the environments, we manually validate them to make sure they behave as expected (see B.8 for details).

B.7 Controller Design

The joints in our robots are controlled by velocity or position controllers. For velocity controllers, we use the built-in inverse dynamics functions in PhysX to compute the balancing forces for a robot. We then apply the internal PD controllers of PhysX by setting stiffness to 0 and damping to a positive constant, where damping is used to drive a robot to a given velocity. We additionally add a first-order low-pass filter, implemented as an exponential moving average, to the input velocity signal, which is a common practice in real robotics systems [78]. Position controllers are built on top of velocity controllers: the input position signal is passed into a PID controller, which outputs a velocity signal for a velocity controller.

B.8 Environment Validation

Environment construction is not complete without testing. After modeling the environment, we need to ensure the environment has the following properties: (1) The environment is correctly modeled with realistic parameters. As the environment contains hundreds of parameters, including the friction coefficients, controller parameters, object scales, etc, its correctness needs comprehensive checking; (2) The environment is solvable. We need to check that the task can be completed within the allocated time steps, and that the physical parameters allow task completion. For example, the weight of a target object to be grasped must be smaller than the maximum force allowed for lifting the robot gripper; (3) The physical simulation is free from significant artifacts. Physical simulation faces the trade-off between stability and speed. When the simulation frequency and the contact solver iterations are too small, simulation artifacts such as jittering and interpenetration can occur; (4) There does not exist undesired exploits and shortcuts.

To inspect our environments, after modeling them, we first use the SAPIEN viewer to visually inspect the appearance of all assets. We also inspect the physical properties of crucial components in several sampled environments. Next, we design a mouse-and-keyboard based robot controller. It controls the robot gripper in Cartesian coordinates using inverse kinematics of the robot arm. We try to manually solve the tasks to identify potential problems in the environments. This manual process can identify most solvability issues and physical artifacts. However, an agent could still learn unexpected exploits, entailing us to iteratively improve the environments: we execute the demonstration collection algorithm on the current environment and record videos of sampled demonstration trajectories. We then watch the videos to identify causes of success and failure, potentially spotting unreasonable behaviors. We finally investigate and improve the environment.

B.9 Manually Processed Collision Shapes

As described in Section 4.1.3, we manually decompose the collision shapes into convex shapes. This manual process is performed on the Bucket objects. We justify this choice in Fig. 6.

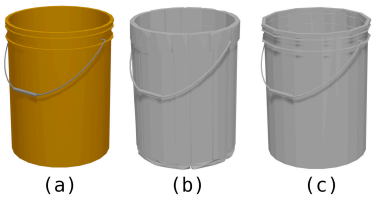

Figure 6: When decomposing a bucket (a), standard VHACD [43] algorithm (b, 2340 faces) misses details, and tends to produce artifacts, such as bumps and seams, that make visual appearances quite different from collision shapes, so we manually process the mesh (c, 1445 faces).

C Details of Demonstration Collection

C.1 Dense Reward Design

A ManiSkill task is defined on different objects of the same category. This natural structure implies an efficient way to automatically construct dense reward functions for all environments of each task. For each task, we manually design a general template, then automatically translate this template into a smooth reward for each environment. The reward templates are also open-sourced.

We use a multi-stage reward template for all of our tasks. In each stage, we guarantee that rewards in the next stage are strictly larger than rewards in the current stage to prevent RL agents from staying in an intermediate stage forever. We also carefully design our reward template at stage-transition states to ensure smoothness of our rewards.

C.1.1 OpenCabinetDoor and OpenCabinetDrawer

For environments in OpenCabinetDoor and OpenCabinetDrawer, our reward template contains three stages. In the first stage, an agent receives rewards from being close to the handle on the target link (door or drawer). To encourage contact with the target link, we penalize the Euclidean distance between the handle and the gripper. When the gripper's distance to the target link is less than a threshold, the agent enters the second stage. In this stage, the agent gets a reward from the opening angle of the door or the opening distance of the drawer. When the agent opens the door or drawer enough, the agent enters the final stage. In the final stage, the agent receives a negative reward based on the speed of the target link to encourage the scene to be static at the end of the trajectory.

C.1.2 PushChair

The reward template for PushChair contains three stages. In the first stage, an agent receives reward from moving towards the chair. To encourage contact with the chair, we compute the distance between the robot end effectors and the chair and take its logarithm as reward. When the robot end effectors are close enough to the chair, the agent enters the second stage. In the second stage, the agent receives rewards based on the distance between the chair's current location and the target location. The agent receives additional rewards based on the angle between the chair's velocity vector and the vector pointing towards the target location. In our experiments, we find that this term is critical. When the chair is close enough to the target location, the agent enters the final stage. In the final stage, the agent is penalized based on the linear and angular velocity of the chair, such that the agent learns to keep the chair static. In all stages, the agent is penalized based on the chair's degree of tilt in order to keep the chair upright.

C.1.3 MoveBucket

The reward template for MoveBucket consists of four stages. In the first stage, an agent receives rewards from moving towards the bucket. To encourage contact with the bucket, we compute the distance between the robot end effectors (grippers) and the bucket and take its log value as reward. When the robot end effectors are close enough to the bucket, the agent enters the second stage. In the second stage, the agent is required to lift the bucket to a specific height. The agent receives a position-based reward and a velocity-based reward that encourage the agent to lift the bucket. In our experiments, we find that it is very difficult for the agent to learn how to lift the bucket without any

domain knowledge. To ease the difficulty, we use the angle between the two vectors pointing from the two grippers to the bucket's center of mass as a reward. This term encourages the agent to place the two grippers on opposite sides of the bucket. We also penalize the agent based on the grippers' height difference in the bucket frame so that the grasp pose is more stable. Once the bucket is lifted high enough, the agent enters the third stage. In this stage, the agent receives a position-based reward and a velocity-based reward that encourages the agent to move the bucket towards the target platform. When the bucket is on top of the platform, the agent enters the final stage, and it is penalized based on the linear and angular velocity of the bucket, such that the agent learns to hold the bucket steadily. In all stages, the agent is also penalized based on the bucket's degree of tilt to keep the bucket upright. Since it is harder to keep the bucket upright in MoveBucket than in PushChair, we take the log value of the bucket's degree of tilt as an additional penalty term so that the reward is more sensitive at near-upright poses.

C.2 Agent Training and Demonstration Collection

# Different Cabinets	1	5	10	20
Success Rate	100%	82%	2%	0%

Table 5: The success rates of SAC [26] agents on OpenCabinetDrawer trained from scratch with 10^6 time-steps on different numbers of cabinets. The SAC agents are trained in the state mode using our designed dense rewards. Jointly training a single RL agent on a large number of environments (objects) from scratch to collect demonstrations is infeasible.

Since different environments of a task contain different objects of the same category, a straightforward idea to collect demonstrations is to train a single agent from scratch directly on all environments through trial-and-error, which seems feasible at first glance. However, as shown in Table 5, even with carefully-designed rewards, the performance of such approach drops sharply as the number of different objects increases.

While directly training one single RL agent on many environments of a task is very challenging, training an agent to solve a single specific environment is feasible and well-studied. Therefore, we collect demonstrations in a divide-and-conquer way: We train a population of SAC [26] agents on a task and ensure that each agent is able to solve a specific environment. These agents are then used to interact with their corresponding environments to generate successful trajectories. In this way, we can generate an arbitrary number of demonstration trajectories.

For all environments, we train our agents for 2.0×10^6 steps. To ensure the quality of our demonstrations, in a few cases where the success rate of an SAC agent is less than 0.3, we retrain the agent. We then uniformly sample the initial states and use the trained agents to collect 300 successful trajectories for each environment of each task.

To speed up the training process, we use 4 processes in parallel to collect samples. For better demonstration quality, during training, we remove the early-done signal when agents succeed. While this potentially lowers the success rates of agents, we find that this leads to more robust policy at near-end states. However, during demonstration collection, we stop at the first success signal. The entire training and demonstration generation process takes about 2 days in total for all environments and all tasks on 4 8-GPU and 64-CPU machines. The detailed hyperparameters of SAC can be found in Table 6.

We would like to note that our benchmark is not a generic learning-from-demonstrations benchmark, so comparing the diversity of our RL-generated demonstrations with human demonstrations is not the focus of our work (we focus on solving the tasks themselves). However, we do empirically observe that our RL-generated demonstrations are diverse, especially given that a large number of demonstrations have been provided. For example, in OpenCabinetDrawer, the demonstration trajectories show two different behaviors: opening the drawer by pulling the handle, or by pulling the side of the drawer out. One reason for such diverse behaviors could come from our reward design: While the multi-stage reward template we designed encourages an agent to grasp the handle of a drawer, we do not explicitly restrict how it opens the drawer. As long as the opening method is feasible and achieves the desired goal state, the trajectory will be considered successful.

Hyperparameters	Value
Optimizer	Adam
Learning rate	3×10^{-4}
Discount (γ)	0.95
Replay buffer size (γ)	10^6
Number of hidden layers (all networks)	3
Number of hidden units per layer	256
Number of threads for collecting samples	4
Number of samples per minibatch	1024
Nonlinearity	ReLU
Target smoothing coefficient(τ)	0.005
Target update interval	1
Gradient steps	1
Total Simulation Steps	2×10^6

Table 6: The hyperparameters of SAC for demonstration generation.

D Implementation Details of Baseline Architectures, Algorithms, and Experiments

D.1 Point Cloud Subsampling

Since point clouds are captured from the 3 cameras mounted on the robot with resolution 400×160, as described in Section 2.3, point clouds without postprocessing (192000 points per frame) would be very memory efficient and significantly reduces the training speed. Therefore, we downsample the raw point cloud by first sampling 50 points for each segmentation mask (if there are fewer than 50 points, then we keep all the points). We then randomly sample from the rest of the points where at least one of the segmentation masks is true, such that the total number of those points is 800 (if there are fewer than 800, then we keep all of them). Finally, we randomly sample from the points where none of the segmentation masks are true and where the points are not on the ground (i.e. have positive z-coordinate value), such that we obtain a total of 1200 points.

For the convenience of researchers and benchmark users, we also made this downsampled point cloud demonstrations dataset public, so users do not need to render the demonstrations locally if they wish to use our subsampling function. Instructions for download can be found in our repository https://github.com/haosulab/ManiSkill-Learn.

D.2 Network Architectures

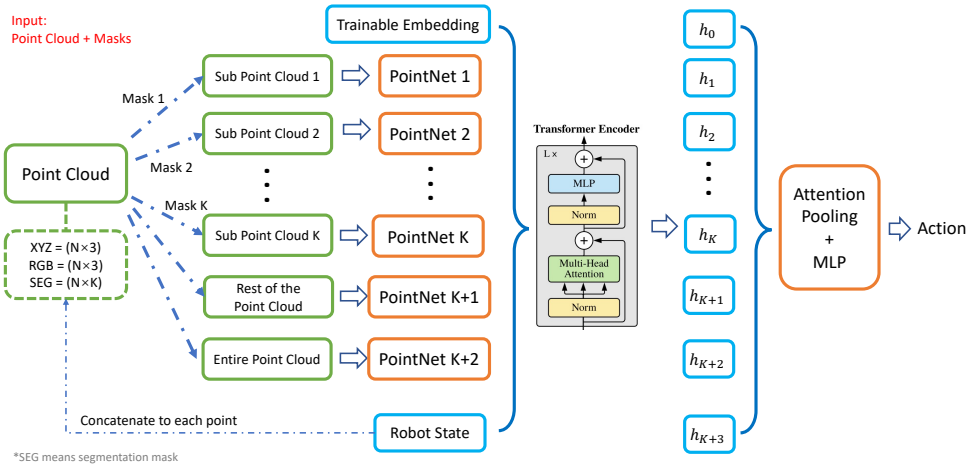

Figure 7: Architecture diagram for our "PointNet + Transformer" model.

For all of our PointNet policy network models, we concatenate the features of each point (which include position, RGB, and segmentation masks) with the *robot state* (as mentioned in Section 2.1 in our main paper) to form new point input features. For the position feature, we first calculate the mean coordinates for the point cloud / sub point cloud, then concatenate it with the original position subtracting the mean. We found such normalized position feature significantly improve performance.

In our vanilla PointNet model, we feed all point features into one single PointNet. The PointNet has hidden layer dimensions [256, 512], and the global feature is passed through an MLP with layer sizes [512, 256, action_dim] to output actions.

For our PointNet + Transformer model, we use different PointNets to process points having different segmentation masks. If the masks have dimension k, then we use $k+2$ PointNets (one for each of the segmentation masks, one for the points without any segmentation mask, and one for the entire point cloud) with hidden dimension 256 to extract $k+2$ global features. We also use an additional MLP to output a 256-d hidden vector for the robot state alone (i.e. the robot state is not only concatenated with the point features and fed into the PointNets, but also processed alone through this MLP). The global point features, the processed robot state vector, and an additional trainable embedding vector (serving as a bias for the task) are fed into a Transformer [67] with $\mathrm{d}_{\mathrm{model}} = 256$ and $\mathrm{d}_{\mathrm{ff}} = 1024$. We did not add position encoding to the Transformer, as we found it significantly hurts performance. The output vectors are passed through a global attention pooling to extract a representation of dimension 256, which is then fed into a final MLP with layer sizes [256, 128, action_dim] to output actions. A diagram of this architecture is presented in Figure 7. All of our models use ReLU activation.

D.3 Implementation Details of Learning-from-Demonstration Algorithms

We benchmark imitation learning with Behavior Cloning (BC), along with two offline-RL algorithms: Batch-Constrained Q-Learning (BCQ) [23] and Twin-Delayed DDPG with Behavior Cloning (TD3+BC) [21]. Different from BC, BCQ does not directly clone the demonstration actions given input, and instead uses a VAE to fit the distribution of actions in the demonstration. It then learns a Q function that estimates the reward of actions given input, and selects an action with the best reward among samples during inference. TD3+BC [21] adds a weighted BC loss to the TD3 loss to constrain the output action to the demonstration data. The original paper also normalizes the features of every state in the demonstration dataset, but this trick is not applicable in our case as our inputs are visual. There are also other offline-RL algorithms like CQL [33], and we leave them for future work.

Hyperparameters	Value
Batch size	64
Perturbation limit Φ	0.00
Action samples n during evaluation	100
Action samples n during training	10
Learning rate	5×10^{-4}
Discount (γ)	0.95
Nonlinearity	ReLU
Target smoothing coefficient(τ)	0.005

Table 7: The hyperparameters of BCQ.

Hyperparameters	Value
α	0.02
Learning rate	5×10^{-4}
Action noise	0.2
Noise clip	0.5
Discount (γ)	0.95
Nonlinearity	ReLU
Target smoothing coefficient(τ)	0.005

Table 8: The hyperparameters of TD3+BC.

For Q-networks in BCQ [23] and TD3+BC [21], when using the PointNet + Transformer model, the action is concatenated with the point features and the state vector and fed into the model. The final feature from the model is fed into an MLP with layer sizes [256, 128, 1] to output Q-values. The

α	0.00	0.02	0.2	2.5
Success Rate	0.85	0.31	0.01	0.00

Table 9: The success rates of TD3+BC trained with different values of α on one environment of OpenCabinetDrawer and 300 demonstration trajectories. The algorithm becomes equivalent to BC if $\alpha = 0$.

VAE encoder and decoder in BCQ uses similar architecture, and the dimension of the latent vector z equals 2 times the action space dimension. The hyperparameters for BCQ and TD3+BC are shown in Table 7 and Table 8.

Note that in TD3+BC, $\pi = \text{argmax}_\pi \mathbb{E}_{(s,a)\sim\mathcal{D}} \left[\lambda Q(s, \pi(s)) - (\pi(s) - a)^2\right]$, where $\lambda = \frac{\alpha}{\frac{1}{N}\sum_{(s_i,a_i)} |Q(s_i,a_i)|}$. In the original paper, $\alpha = 2.5$, and the algorithm is equivalent to BC if $\alpha = 0$. Interestingly, as shown in Table 9, we found that when α is non-zero, the performance of TD3+BC is always worse than BC, even when α has decreased 100 times from the value in the original paper. However, in our previously reported results, we used $\alpha = 0.02$ to illustrate the performance comparison between TD3+BC and BC, since setting $\alpha = 0$ is not interesting, and does not distinguish TD3+BC from BC.