A IMPLEMENTATION OF OUR 3D METHOD

2D encoder. We use ResNet18 as the encoder.

3D encoder consists of two 3D transposed convolutions activated by Leaky ReLU function.

3D decoder first reshapes the deep voxel $(C/D) \times D \times H \times W$ back to $C \times H \times W$, and consists of one 2D 1×1 convolution and three transposed convolutions to reconstruct the RGB images.

PoseNet concatenates two images as the input and reduces the dimension to 6 for predicting Euler angles $[\alpha, \beta, \gamma]^\top$ and the translation $[x, y, z]^\top$, by seven 2D convolutions.

3D transformation. We warp the grid such that the voxel at location $p = [i, j, k]^\top$ will be warped to $\hat{p}$, which is computed as

$$\hat{p} = Rp + t \tag{4}$$

where R, t is the 3×3 rotation matrix and translation vector corresponding to the camera pose. In our implementation, the warp is performed inversely and the value at fractional grid location is trilinearly sampled. In addition, since there exists misaligned voxels during the sampling procedure caused by the coarse deep voxel representation, we apply two 3D convolutions to refine and correct these mismatches.

3D pretraining. We randomly select 20 classes in CO3D and sample I_{src} and I_{tgt} with a bounded interval $b = 9$. We train $\sim 250k$ iterations with batch size 32.

3D finetuning. In the finetuning phase, we apply a less frequent update when doing the 3D task, *i.e.*, performing λ_{up} 3D update every 1 RL update. In practice, we set $\lambda_{\text{ft}} = 10^{-2}$, $\text{lr}_{\text{RL}} = 10^{-3}$, and $\lambda_{\text{up}} = 0.5$. In addition, we only apply the reconstruction loss.

B BASELINES

From scratch, also called vanilla SAC, does not use any pretrained model and utilizes a 2D encoder with fourteen 2D convolutions activated by ReLU function. Our implementation generally follows DrQ (Kostrikov et al., 2020). The actor consists of fully connected layers activated by Tanh function. The critic applys fully connected layers activated by ReLU function and predicts double action value funtions Q with a shared encoder and two different heads. We apply the same data augmentation as in (Jangir et al., 2022) for better sim-to-real transferring, including *random shift* and *color jitter*.

ImageNet. We replace the 2D encoder in vanilla SAC with ResNet18 pretrained with supervised learning on ImageNet, to gain a stronger 2D representation.

MoCo. We replace the 2D encoder in vanilla SAC with ResNet18 which is trained by MoCo (v2) (Chen et al., 2020) on ImageNet under the setting where the batch size is 256, the number of epochs is 100, and the initial learning rate is 0.03.

Remove ImageNet normalization for usage. Our baseline methods MoCo and ImageNet are both pretrained with ImageNet and all input images are preprocessed with the normalization of ImageNet, *i.e*, with the mean $(0.485, 0.456, 0406)$ and the standard deviation $(0.229, 0.224, 0.225)$. A natural way to apply such pretrained networks in RL is using the same normalization to maintain the representation ability of the pretrained networks. However, by empirical experiments we find that normalizing the images directly into $[0, 1]^d$ achieves a much better performance, as shown in Figure 9a. Thus we adopt a stronger version as our baseline.

C DESIGN OF CAMERAS

Design of the static view. The static view is generally used for all baselines and our algorithm, for both the training phase and the inference phase. Thus the inner requirement is that the static view should contain the majority of useful information for the robotic task, to gain a strong baseline. In practise we carefully select a unified static view for all tasks.

Design of the dynamic view. The dynamic camera that shots I_{tgt} is essential, which largely decides whether the image reconstruction could work. Based on the priori that our task is object-centric and

the intuition that interaction between the robot and the object is our focus, we move the dynamic view in a object-centric manner, *i.e.*, moving along a circle around the center of the scene, starting from the static view as a initial position. The center of the scene is designed to cover the necessary objects and the main scene information. For example, in the *peg in box* task, where the robot is required to move the peg into the box on the table, the box is essential to understand and solve this task, and thus the box could be seen as the center of the scene.

Formally, let $[x_s, y_s, z_s]^\top$ denote the position of the static view, $[x_d, y_d, z_d]^\top$ denote the position of the dynamic view, $[x_c, y_c, z_c]^\top$ denote the center of the scene, r denote the radius of the circle, ϕ_s denote the rotation angle of the static view, and ϕ_d denote the rotation angle of the dynamic view from the static view. We also introduce ϕ, which denotes the range of the rotation of the dynamic view. Then the translation of the cameras is given in Equation 5 and 6. The rotation angle is automatically computed by making the camera point to the center of the scene with the z-axis in the plane perpendicular to the ground.

$$[x_s, y_s, z_s]^\top = [x_c, y_c, z_c]^\top + r \cdot [\sin\phi_s, \cos\phi_s, 0]^\top, \text{ where } \phi_s \text{ is predefined.} \tag{5}$$

$$[x_d, y_d, z_d]^\top = [x_c, y_c, z_c]^\top + r \cdot [\sin(\phi_s + \phi_d), \cos(\phi_s + \phi_d), 0]^\top, \text{ where } \phi_d \in [0, \phi]\,. \tag{6}$$

D Novel View Synthesis in Real

Videos consisting of synthesised views are displayed in our project website https://3d4rl.github.io/. In this section we describe details of how we generate the synthesised views for the real world.

Let I_{real} denote the image shot in the real world from the same static view as in simulation. The deep voxel representation is thus generated as $g_\theta(f_\theta(I_{\text{real}}))$. Since we only have one static camera in the real world (we could have other cameras in real, but it is not necessary for our method), we use the simulation images $I_{\text{src}}, I_{\text{tgt}}$ (the same notation as before) to predict the relative transformation by PoseNet and get transformation R, t. Then the reconstructed image is $\hat{I}_{\text{recon}} = h_\theta(T_{R,t}(g_\theta(f_\theta(I_{\text{real}}))))$, which should be in the same view as I_{tgt}, but different in the scene content.

To generate videos, we do interpolation on the generated transformations and apply these new transformations to get interpolated views,. The output of PoseNet is $[\alpha, \beta, \gamma, x, y, z]^\top$, and let $\lambda \in [0, 1]$ denote the interpolation factor. Then the interpolated transformations are

$$(1-\lambda)[0,0,0,0,0,0]^\top + \lambda[\alpha, \beta, \gamma, x, y, z]^\top = [\lambda\alpha, \lambda\beta, \lambda\gamma, \lambda x, \lambda y, \lambda z]^\top, \tag{7}$$

from which we could trivially gain R, t.

E Additional Results

E.1 Dynamic Camera Moving Range ϕ

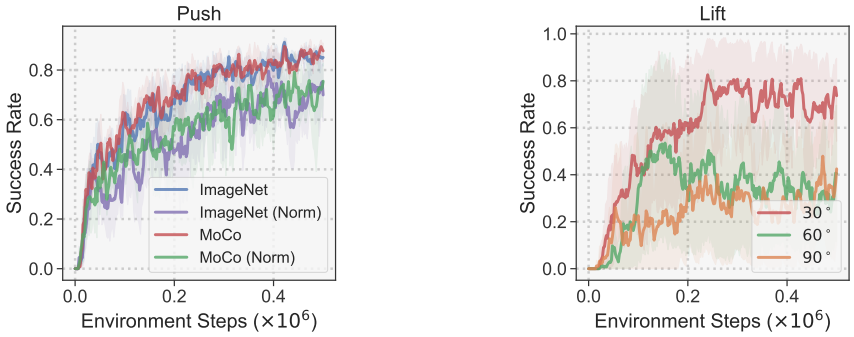
The moving range of the dynamic camera affects the performance of our 3D algorithm. Specifically, a larger ϕ may impede the learning process and hurt the performance. We use *lift* task as an example to illustrate the effect of ϕ, as shown in Figure 9b. We find that with a relatively small range, *i.e.*, 30°, 3D could be more stable.

E.2 ImageNet Normalization in Baselines

When the visual representation is frozen, the ImageNet normalization is generally used across all pretrain methods (Xiao et al., 2022; Nair et al., 2022; Parisi et al., 2022). However, we find that when we could train the policy and the visual representation end-to-end, it would be better to not apply the ImageNet normalization, as shown in Figure 9a. We thus adopt the stronger baseline.

E.3 Pose Estimation

Our 3D method trains a PoseNet that could estimate the relative pose between two frames and we evaluate our pose estimation results quantitatively in this section. For a whole trajectory generated



(a) (b)

Figure 9: *(a)* Compare baselines with and without ImageNet Normalization. *(b)* Success rate of our method with different camera moving range ϕ on *lift*.

Table 4: **Quality of pose estimation for peg in box task.** Each table shows the root mean square error (RMSE) and the maximal error (MaxE) of different dynamic camera angle ϕ_d given certain finetuning scale λ_{ft}. We only train with $\phi_d = 30°$. We could observe that the finetuning leads to consistent smaller errors.

Pretrain			$\lambda_{\text{ft}} = 0.01$			$\lambda_{\text{ft}} = 0.10$			$\lambda_{\text{ft}} = 1.00$		
ϕ_d	RMSE↓	MaxE↓	ϕ_d	RMSE↓	MaxE↓	ϕ_d	RMSE↓	MaxE↓	ϕ_d	RMSE↓	MaxE↓
15	0.041	0.093	15	0.041	0.089	15	0.040	0.091	15	0.046	0.126
30	0.066	0.142	30	0.059	0.122	30	0.033	0.097	30	0.041	0.124
45	0.120	0.246	45	0.064	0.150	45	0.046	0.102	45	0.044	0.120
60	0.174	0.334	60	0.130	0.393	60	0.132	0.382	60	0.133	0.365
avg	0.100	0.204	avg	0.073	0.189	avg	0.063	**0.168**	avg	**0.066**	0.184

by the interaction between our agent and the environment, we estimate the relative pose between the dynamic camera and the static camera for each timestep. Since the estimated transformation is in the coordinate space of deep voxels, Umeyama alignment (Umeyama, 1991) is applied to align the predicted trajectory with the ground truth trajectory provided by our simulation environment. We set diverse dynamic camera angles to test both in-domain $(15°, 30°)$ and out-of-domain $(45°, 60°)$ pose estimation under various finetuning scales. Results in Table 4 show that our method reduces the pose estimation error compared to the network that is only pretrained with CO3D dataset. Our method could also generalize to 45 degrees with a small error equal to 0.064, nearly half of the one with only pretraining. In addition, we find that larger finetuing scales generally reduce the error, and even finetuning with a very small scale could result in a gap compared to the pretrain model.

E.4 Compare with other 3D Pretrain Methods

In our main sections we demonstrate the advantage of our method over 2D representations, and we are now showing that our 3D self-supervised representation is also better than other straightforward pretrain methods that contain 3D information. Specifically, we consider the ResNet50 model pretrained by the depth estimation task using convolutional spatial propagation network (CSPN) (Cheng et al., 2018b). We still freeze the visual representation across methods. The results are

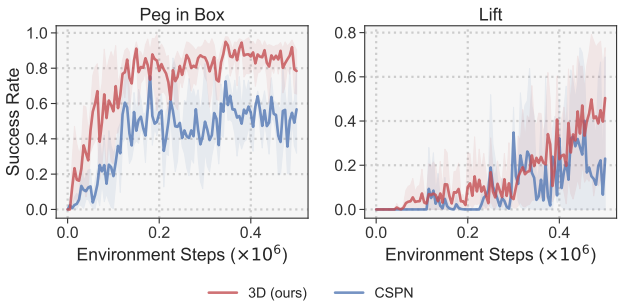


Figure 10: **Success rate of different frozen visual representations.** We compare our 3D visual representation with CSPN (Cheng et al., 2018b) on *peg in box* and *lift*.

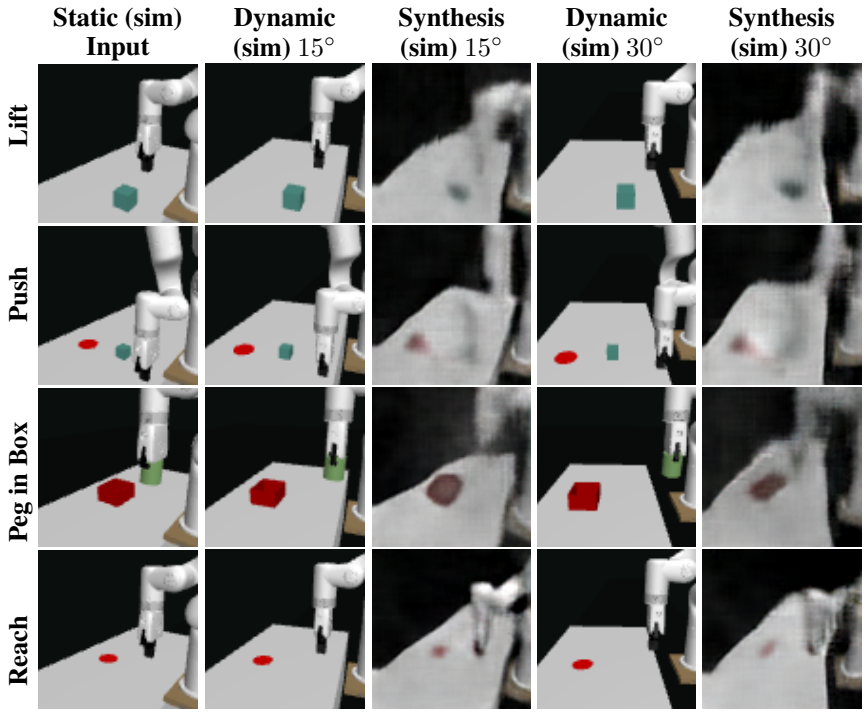


Figure 11: **Novel view synthesis in simulation.** We display the reconstruction results for $\phi_d = 15°, 30°$ in four tasks.

shown in Figure 10 under the same setting as Figure 6. We find that our 3D representation is consistently better on *peg in box* and *lift*, while the CSPN model could also gain reasonable accuracy.

E.5 Computational Overhead

Although our 3D based algorithm is elegantly designed for better sample efficiency, the computational overhead of utilizing the auxiliary task for joint optimization is non-negligible. We measure the computation time for one 3D update (0.038s) and one RL update (0.063s) averaged over 10 iterations on a NVIDIA GeForce RTX 3090. The large overhead is mainly because our method reconstructs the image from the 3D scene latent, which is higher dimensional ($\mathcal{O}(n^3)$) than common 2D methods ($\mathcal{O}(n^2)$). How to make the utilization of 3D information more computational efficient is interesting to explore in our future work.

E.6 Novel View Synthesis Results in Sim and Real

We provide qualitative results of our novel view synthesis both in simulation and in the real world. In Figure 7 we show the synthesis using real world images and our model is only trained in simulation. Figure 11 gives more results in simulation. We also compare the synthesis generated by the pretrained model and the finetuned model in Figure 12, where we find that the pretrained model could grasp the main objects in the scene while the domain gap, *e.g.*, color, could be clearly observed.

F Visualization of our Environments

We give more visualization of our environments, including four xArm environments: *Lift*, *Push*, *Peg in Box*, and *Reach*, and four Meta-World environments: *Basketball*, *Box Close*, and *Coffee Push*, and *Hammer*, shown in Figure 13. For one single xArm task, we are giving different initialization setting, showing the randomization in our environments for generalization. For one single Meta-World task, we show different views along the trajectory of the dynamic camera. We also visualize the perturbed

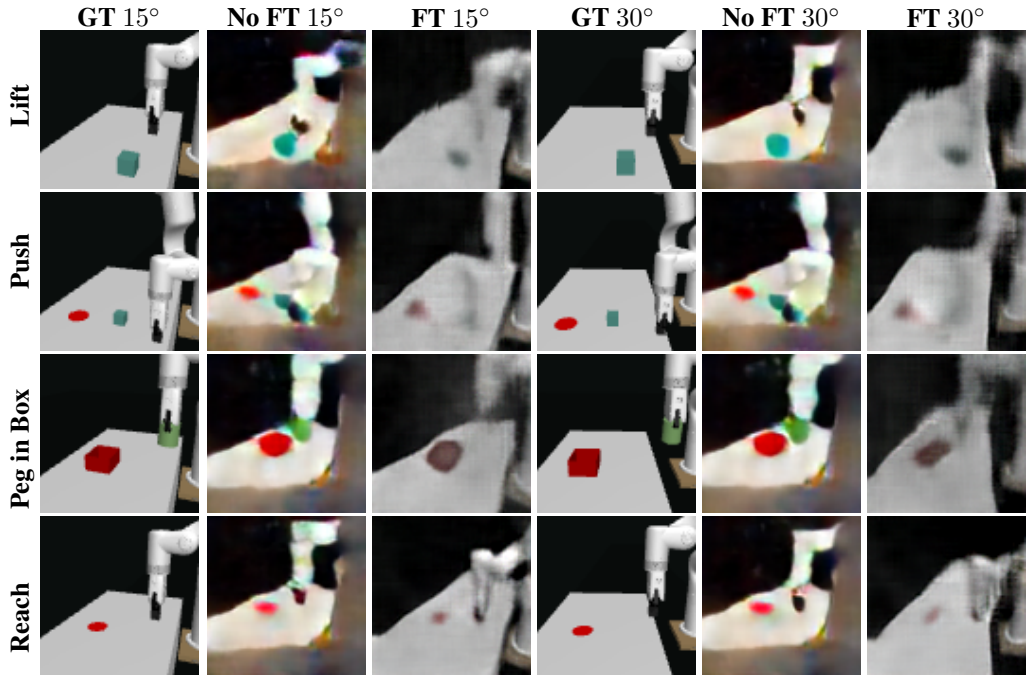


Figure 12: **The effect of 3D finetuning for novel view synthesis in simulation.** *GT* represents for ground truth and *FT* represents for finetuning. We display the reconstruction results for $\phi_d = 15°, 30°$ in four tasks.

simulation environments as shown in Figure 14 and the examples of successful trajectories as shown in Figure 15.

G HYPER-PARAMETERS

We provide all relevant hyper-parameters used in our experiments in Table 5, including both parameters that are discussed and not discussed in our paper.

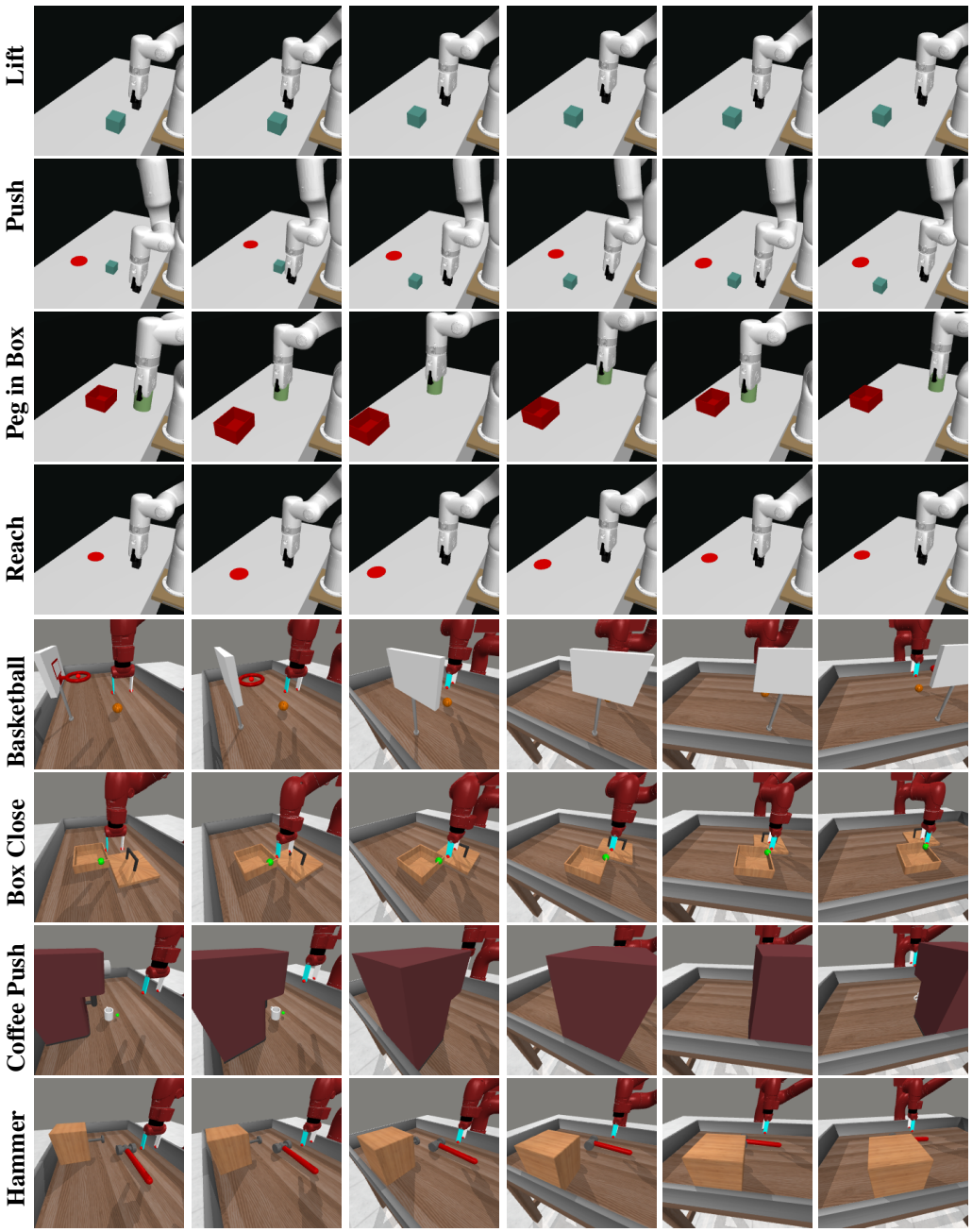


Figure 13: **Visualizations.** We visualize four xArm environments and selected four Meta-World environments. Our xArm environments are shown across different initialization, where initial position of end-effector and objects are randomized. Meta-World environments are shown along the trajectory of the dynamic camera.

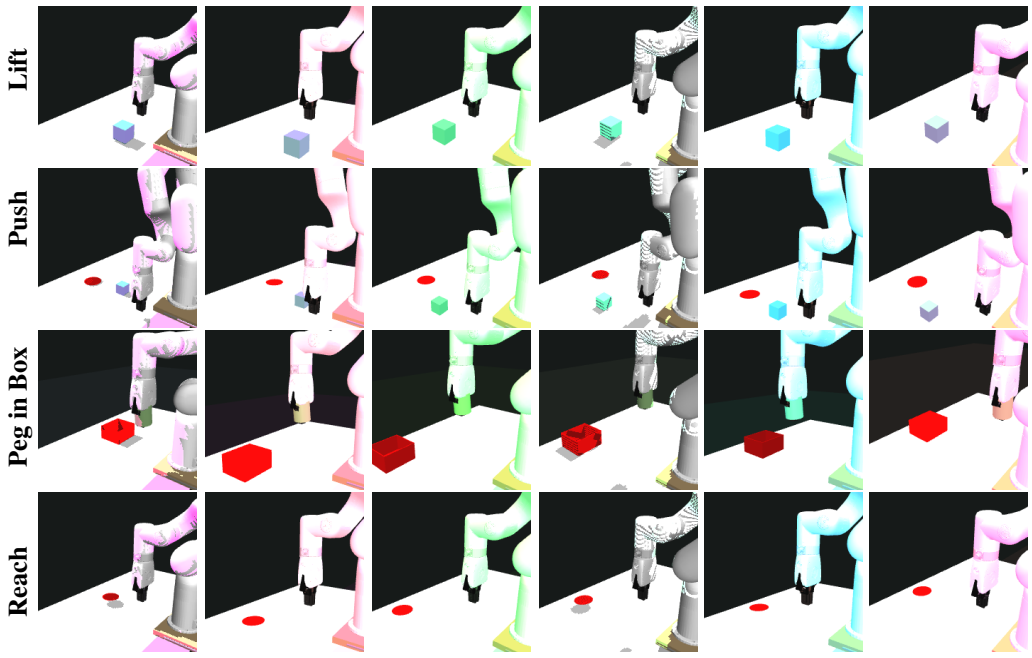


Figure 14: **Visualizations of perturbed simulated environments.** We visualize four xArm environments. Our xArm environments are shown across different initialization, together with texture randomization, lighting randomization, and camera perturbation.

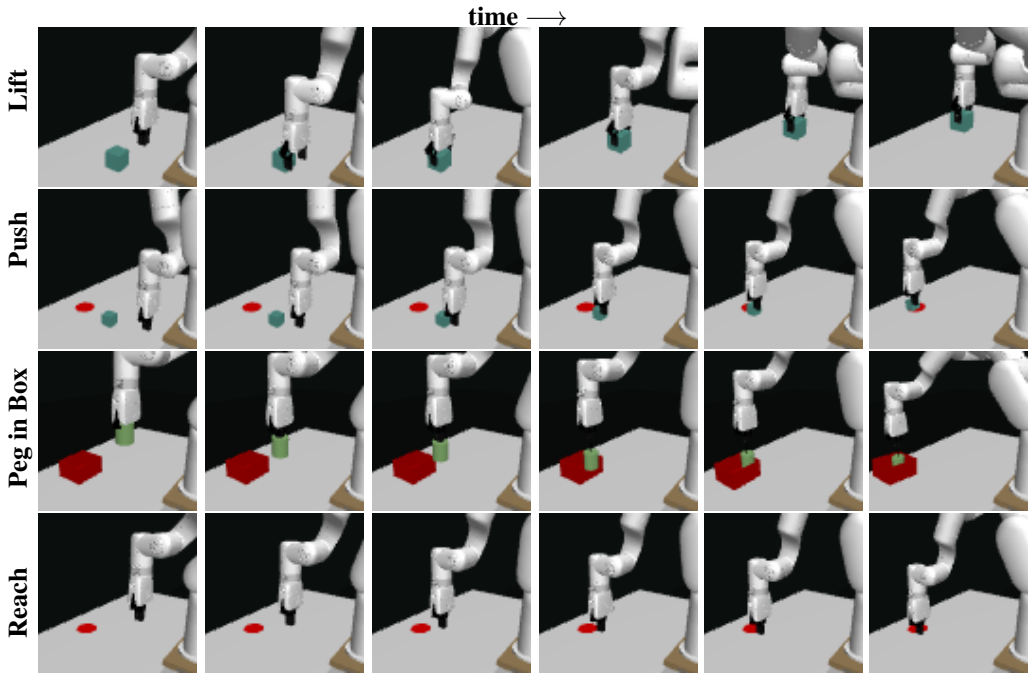


Figure 15: **Visualizations of trajectories in simulated environments.** We sample one successful trajectory for each xArm environment.

Table 5: **Hyper-parameters.**

Variable	Description	Value
b	the bounded interval of sampling training frames	9
lr_{RL}	learning rate of the RL agent	10^{-3}
λ_{up}	frequency of 3D update	0.5
λ_{ft}	finetuning scale	0.01
ϕ	dynamic camera angle range	30°
–	observation shape	$84 \times 84 \times 3$
–	episode length of xArm tasks	50
–	episode length of MetaWorld tasks	200
–	replay buffer capacity	$500k$
–	batch size of replay buffer sampling	128
–	training steps (xArm)	$500k$
–	training steps (Meta-World)	1m
–	discount factor	0.99
–	initial random steps	1000
–	initial temperature	0.1
–	frequency of RL update	1
–	random shift padding	4
–	brightness of color jitter	0.4
–	saturation of color jitter	0.4
–	contrast of color jitter	0.4
–	hue of color jitter	0.5