Appendix for
Multi-Object Manipulation via Object-Centric Neural Scattering Functions

Stephen Tian[1*] Yancheng Cai[1*] Hong-Xing Yu[1] Sergey Zakharov [2]
Katherine Liu[2] Adrien Gaidon[2] Yunzhu Li [1] Jiajun Wu[1]

[1]Stanford University [2] Toyota Research Institute

1. Top-down approach baseline

We evaluate a top-down approach baseline by substituting a PoseCNN pre-trained on a subset of YCB objects to directly regress the pose in place of the perception module of our pipeline. Over 30 trials in the three object manipulation setting, our method achieves a final object position error of $0.078m \pm 0.0056$ while the PoseCNN version achieves $0.093m \pm 0.0056$, where the errors represent standard error. This experimental setup is slightly different from that described in Section 5.2 because the set of objects is further restricted to those that PoseCNN was pre-trained on. While PoseCNN often pro- duces reasonable pose estimates, it frequently incurs large errors in orientation and false detections.

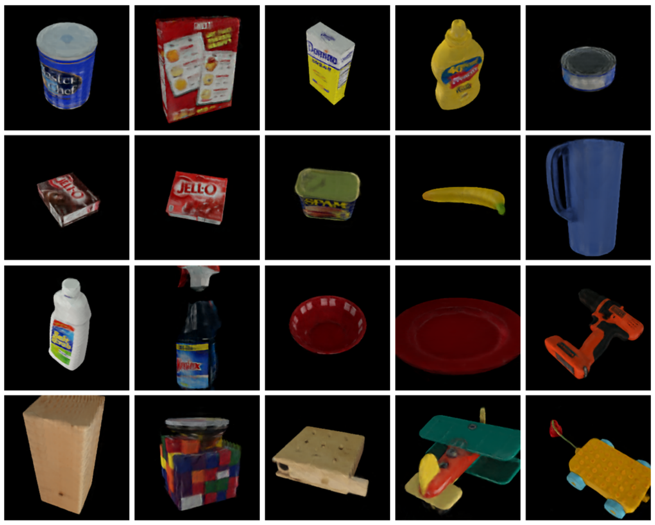

Figure 1. The 20 YCB objects used in the simulated experiments, rendered with KiloOSF.

2. KiloOSF Training Details

We use the implementation of KiloOSFs implemented by Gao *et al.* [3], which is in turn based on the original implementation of KiloNeRF [8]. We use identical hyperparameters for NeRF training as the NSVF objects from the original KiloNeRF implementation (see an example here), except during the pretraining phase, we use `num_samples_per_ray` of 64, and train for 100000 iterations as opposed to 600000 as we find that this is sufficient for our experiments.

For each object, the training data consists of 1000 simulated images with 256×256 resolution, each with a randomized camera viewpoint and light pose. For each object, the camera points at the center of its axis-aligned bounding box. For each image, the camera yaw is uniformly drawn from $[0, 2\pi]$, pitch is uniformly drawn from $[-\pi/2, 0]$, and the distance of the camera to the bounding box center is drawn uniformly from $[0.25, 0.4]$ (meters). The roll is always set to 0. The light source always points at the origin. The light position is randomized by drawing a vector uniformly from $[-20, 20], [-20, 20], [5, 10]$ and normalizing.

Training was conducted on NVIDIA TITAN RTX and RTX 3090 GPUs.

3. Inverse Parameter Estimation Details

We use CMA-ES [4] to perform inverse parameter estimation. Specifically, we leverage the implementation from the open-source Python library `cmaes` [9]. For all experiments, we render images at and perform optimization using 128×128

[*]indicates equal contribution. YC is affiliated with Fudan University; this work was done while he was a summer intern at Stanford.

resolution.

3.1. Object Pose Estimation

We initialize object pose estimation as as a $7 * \texttt{num\_objects}$ dimensional optimization problem, with each 7D vector representing the position and unnormalized quaternion of a single object. Each object position is constrained to be in $[-0.5, 0.5], [-0.5, 0.5], [0, 0.1]$, and each dimension of the quaternion is constrained to be in $[-0.3, 0.3]$ before being normalized to a unit quaternion.

For the CMA parameters, we use the following settings for each experiment:

- **Object masks for all frames (**2 **and** 4 **object settings):** For initial pose estimation at the first planning step, we initialize $\sigma = 0.2$ and use a population size of 500 for 10 iterations. For the following steps, we initialize CMA using the best parameters from the previous step and use $\sigma = 0.01$, using a population size of 50 for 2 CMA iterations.

- **Object masks in only initial frame (**3 **object setting):** For initial pose estimation at the first planning step, we initialize $\sigma = 0.2$ and use a population size of 1000 for 20 iterations. For the following steps, we initialize CMA using the best parameters from the previous step and use $\sigma = 0.01$, using a population size of 500 for 2 CMA iterations.

  We find that these perform similarly, but the latter is more computationally expensive as the number of optimization samples required to optimize the appearance-based MSE loss is higher.

3.2. Light Pose Estimation

For light pose estimation, we initialize the CMA optimizer with bounds $[-1, 1], [-1, 1], [0, 1]$, and then perform 3 iterations of CMA with population size 50, starting from $\sigma = 0.8$.

4. Dynamics Model Training Details and Architecture

Our dynamics model is based on PropNet [6] and contains several modules. It is shown in Figure 3. We detail the architecture of these modules below:

- **Node Encoder**: A three-layer MLP with hidden sizes $[128, 128]$, with LeakyReLU activations after each layer. In addition to the corresponding object pose, for every object the node encoder receives as input the action of the cylindrical pusher object, along with a binary indicator of whether or not the object is the pusher, and two 3D coordinates describing the maximum and minimum of xyz coordinates of the object bounding box vertices, as determined by KiloOSF occupancy.

- **Edge Encoder**: A three-layer MLP with hidden sizes $[128, 128]$, with LeakyReLU activations after each layer.

- **Edge Propagator**: A single linear layer with output size 128 and a LeakyReLU activation.

- **Node Propagator**: A single linear layer with output size 128, followed by a residual connection and then a LeakyReLU activation.

- **Node Predictor**: A two-layer MLP with hidden size 128 and a single LeakyReLU activation after the first linear transformation.

We generate edges based on whether the distance between objects is larger than the largest axis of the axis-aligned bounding box of both objects plus $\epsilon = 0.03$. We propagate edge effects as described in [6] for 3 iterations. While the pose of the pushing cylinder is input into the model, we do not use the model's prediction for its next pose and instead directly compute its future pose based on the action by integrating the specified velocity over the action duration.

Explicitly, to compute the loss for training the 3-step dynamics model given ground truth object states and actions from four consecutive timesteps $s^1, a^1, s^2, a^2, s^3, a^3, s^4$, we roll out the dynamics model autoregressively to predict $\hat{s}^2, \hat{s}^3, \hat{s}^4$. The loss is then $\text{MSE}(s^2, \hat{s}^2) + \text{MSE}(s^3, \hat{s}^3) + \text{MSE}(s^4, \hat{s}^4)$.

When training, we use the Adam optimizer [5] with a learning rate of $3e^{-4}$ and batch size of 32. We perform multi-step prediction by autoregressively rolling out model predictions for 3 steps before calculating the loss averaged over those steps. This makes the model more resistant to accumulating errors when performing prediction for multiple steps.

5. Model-Predictive Control Implementation Details

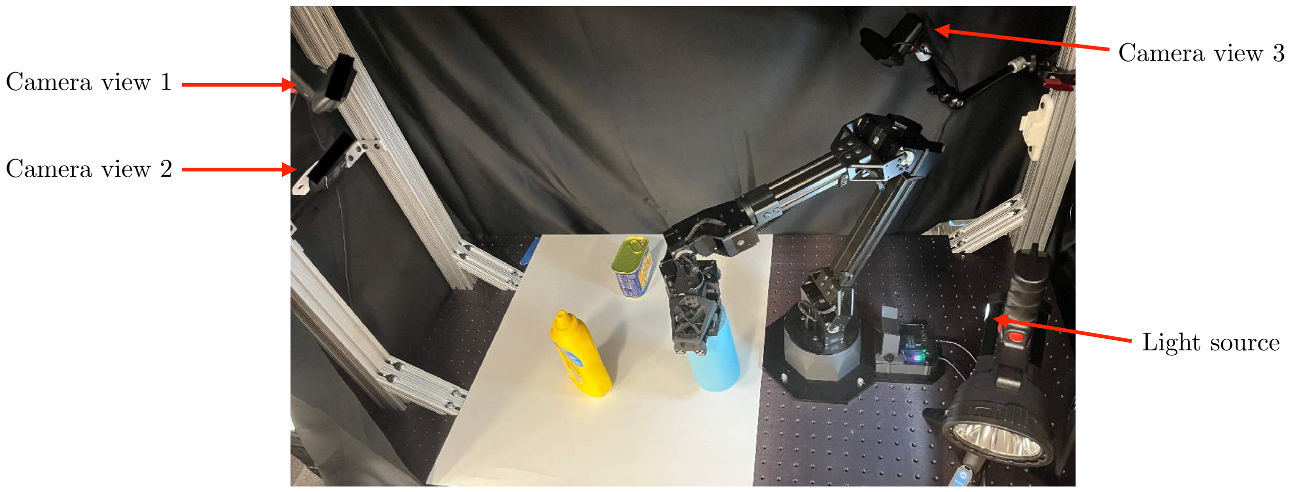

Figure 2. Real robot setup.

We use model-predictive path integral (MPPI) [10] to plan action sequences. We plan using a horizon of 30 steps, setting the MPPI parameter $\gamma = 10$. For sampling, we draw 800 actions in the 3 object settings and 400 actions for the 2 and 4 object settings using a correlated noise sampler (introduced and described in detail in Nagabandi *et al.* [7]) that samples Gaussian noise from $\mathcal{N}(\mu = 0, \sigma^2 = 0.04)$ independently across action dimensions, but with a correlation coefficient $\beta = 0.5$ between the action at each step and the previously sampled step. Increasing the number of action samples only improves the planning performance and we find that values over 400 seem to be satisfactory for our tasks.

We perform replanning after each action taken in the environment, and warm-start the mean of action samples at the next step using the remaining plans from the previous step.

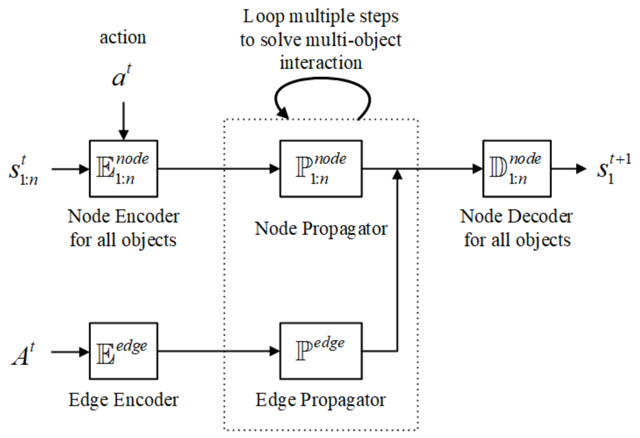
action
a^t
Loop multiple steps to solve multi-object interaction
$s^t_{1:n}$
$\mathbb{E}^{node}_{1:n}$
Node Encoder for all objects
$\mathbb{P}^{node}_{1:n}$
Node Propagator
$\mathbb{D}^{node}_{1:n}$
Node Decoder for all objects
s^{t+1}_1
A^t
$\mathbb{E}^{edge}$
Edge Encoder
$\mathbb{P}^{edge}$
Edge Propagator

Figure 3. The structure of the GNN dynamics model. Within a time step, we loop over propagators multiple times to address multi-step force transfer when more than two objects interact.

6. Real Robot Demo

Please refer to the supplementary video for a demonstration of our method on a real robot.

We use a ViperX300 6-DoF robot arm with the end-effector modified to support a cylidrical pusher as in our simulated experiments. We use OSFs trained on simulated data, and real YCB objects as the experimental objects. We cover the robot tabletop with curtains to reduce the influence of external light, and light the area with a rectangular BIGSUN Q953 spotlight. We record visual observations using 3 Logitech C922 Pro webcams, at 640×480 resolution. Our setup is shown in Figure 2.

We perform camera calibration by computing relative camera poses by collecting a series of images from varying camera poses, and applying COLMAP. We then determine the camera-to-world transform by solving the PnP problem for a particular viewpoint.

For the real robot demo, we perform open-loop control, that is, we plan a sequence of control actions based on the initial estimates of lighting and object pose, and directly roll out the action sequence using the robot.

7. Lighting & Pose Estimation with Real Images

For real-world lighting and pose estimation, we take images using the physical setup described above. We resize all images to 128×128 before performing inverse parameter estimation, and render KiloOSF images during the optimization procedure at 128×128 as well.

For lighting estimation, we parameterize the light as a 7D vector containing the light position, look at point, and intensity. We set the CMA optimizer to optimize scaled versions of these values ($*10$ for the light position and look at point, and $*3/20$ for the intensity) and then optimize with bounds $[-20, 20]$, $[-20, 20]$, $[0, 40]$, $[-20, 20]$, $[-20, 20]$, $[-20, 20]$, $[0, 20]$. This is to encourage the variances of each of the optimization variables to be at a similar scale. Optimization is initialized at (scaled) values $[0, 0, 20, 0, 0, 0, 15]$. We perform 50 CMA-ES iterations at a population size of 15, with initial $\sigma = 7.5$.

For object pose estimation, we use the same parameterization as described previously. We initialize the object pose estimates with estimates from the pretrained PoseCNN provided here. We then use 125 CMA-ES iterations with population size 15 and initial $\sigma = 0.05$. The loss is the summed intersection-over-union (IOU) of the provided object masks and masks computed during compositional KiloOSF rendering, summed over objects and viewpoints.

8. Environment and Data Collection

Our simulated environment is built using PyBullet [2]. We use a time step of $50/240$ as the length of each action. During random data collection, we randomly initialize the position of 3 objects between x and y positions of $[-0.15, 0.15]$, $[-0.15, 0.15]$ and a random z-axis rotation between $[0, 2\pi]$, and allow them to fall to the surface of the table by waiting for 1000 simulation steps before the trajectory begins. Then we sample 50 steps of random uniform velocities for the pusher in $[-0.3, 0.3]$, $[-0.3, 0.3]$.

We use a subset of 18 YCB objects [1], that are enumerated below. We select objects that are larger, rigid, opaque, and non-articulated.

1. 002_master_chef_can
2. 003_cracker_box
3. 004_sugar_box
4. 006_mustard_bottle
5. 007_tuna_fish_can
6. 008_pudding_box
7. 009_gelatin_box
8. 010_potted_meat_can
9. 011_banana
10. 019_pitcher_base
11. 021_bleach_cleanser
12. 022_windex_bottle
13. 024_bowl
14. 029_plate
15. 036_wood_block
16. 071_nine_hole_peg_test
17. 072a_toy_airplane
18. 073g_lego_duplo

9. Computational Cost

The training time for KiloOSF for a single object is about 7 hours wall clock time using a single NVIDIA TITAN RTX GPU. During inference (control) time, performing a single compositional render is on the order of magnitude of 100ms for a three-object scene. As a result the initial pose estimation is on the order of 30 minutes of wall-clock time, and subsequent pose updates 3-5 minutes of wall-clock time. Combining our method with a top-down approach described previously has the potential to significantly improve sample efficiency, and is an exciting direction for future work.

References

[1] Berk Calli, Arjun Singh, Aaron Walsman, Siddhartha Srinivasa, Pieter Abbeel, and Aaron M Dollar. The YCB object and model set: Towards common benchmarks for manipulation research. In *2015 international conference on advanced robotics (ICAR)*, pages 510–517. IEEE, 2015. 4

[2] Erwin Coumans and Yunfei Bai. PyBullet, a python module for physics simulation for games, robotics and machine learning. 2016. 4

[3] Ruohan Gao, Zilin Si, Yen-Yu Chang, Samuel Clarke, Jeannette Bohg, Li Fei-Fei, Wenzhen Yuan, and Jiajun Wu. Objectfolder 2.0: A multisensory object dataset for sim2real transfer. In *CVPR*, 2022. 1

[4] Nikolaus Hansen, Sibylle D. Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary Computation*, 11(1):1–18, 2003. 1

[5] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 2

[6] Yunzhu Li, Jiajun Wu, Jun-Yan Zhu, Joshua B Tenenbaum, Antonio Torralba, and Russ Tedrake. Propagation networks for model-based control under partial observation. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 1205–1211. IEEE, 2019. 2

[7] Anusha Nagabandi, Kurt Konolige, Sergey Levine, and Vikash Kumar. Deep dynamics models for learning dexterous manipulation. In *Conference on Robot Learning*, pages 1101–1112. PMLR, 2020. 3

[8] Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. KiloNeRF: Speeding up neural radiance fields with thousands of tiny mlps. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 14335–14345, 2021. 1

[9] Masashi. Shibata. Cma-es. `https://github.com/CyberAgentAILab/cmaes`, 2022. 1

[10] Grady Williams, Andrew Aldrich, and Evangelos Theodorou. Model predictive path integral control using covariance variable importance sampling. *arXiv preprint arXiv:1509.01149*, 2015. 3