Evo-NeRF: Evolving NeRF for Sequential Robot Grasping ※ Supplemental Material ※

Anonymous Author(s)
Affiliation
Address
email

A Evo-NeRF

A.1 NeRF and scene parameters

We use the default Instant-NGP parameters except for the following changes: (1) we use a learning rate of 0.02 instead of 0.01 (2) we use a hash table size of 2^{17} feature vectors instead of 2^{19} (3) we use a density network with 16 hidden neurons instead of 64. These changes made small improvements in geometry learning speed. We also increase the frequency of extrinsic optimization gradient steps (n_steps_between_cam_updates parameter) to every step, improving the speed of convergence on noisy camera poses added during arm motion. The scene bounds ("aabb_scale") are set to a 2 meter cube to fit the entire workspace inside, with a scene scale of 1.0. We set the near distance for raymarching during NeRF training to be 0, to avoid missing objects if they are close to the camera.

A.2 Capture trajectory

The full capture trajectory is centered at the center of the workspace, with θ values ranging from $85°$ to $75°$ (θ rotates about the z axis upwards from the table, such that x points away from the robot). The ϕ range, which describes inclination from the table surface, goes from $15°$ to $50°$. The arm makes 3 sweeps about the z axis, linearly varying the ϕ value between the range on each sweep, as visualized by the red arrow in Fig. 1. We use 1280x720 images, with a whitebalance and exposure which are held static after an auto-calibration from the camera.

A.3 Evo-NeRF parameters

For TV-regularization, we sample $N = 256000$ points at each iteration with a rejection sampling threshold of 0.01 for minimum local density. The sampling radius r we use is 0.3mm, and the loss scaling λ_{tv} is $\frac{15}{N}$. TV-loss is implemented as a set of CUDA kernels for speed, resulting in only about a 10% slowdown of training. To implement coarser ray sampling, the value of the parameter which controls sample acceleration, ("cone_angle_constant" in Instant-NGP) is 0.04, up from the default value of 0.004.

B Dataset generation

We choose the 7 object meshes based on the three criteria: (1) likely to be made of glass, (b) fit within the workspace of YuMi, (c) has a watertight mesh with outward-facing surface normals.

For grasp generation, we calculate the stable pose orientations of each mesh and rank them by their quasi-static probabilities using Trimesh [1]. Based on the ranking, we select the top 10 stable poses to sample 1000 grasps. We ignore stable poses where no grasp exists. We analytically calculate grasp success via robust wrench resistance [2]. We perturb the grasp pose with small translation

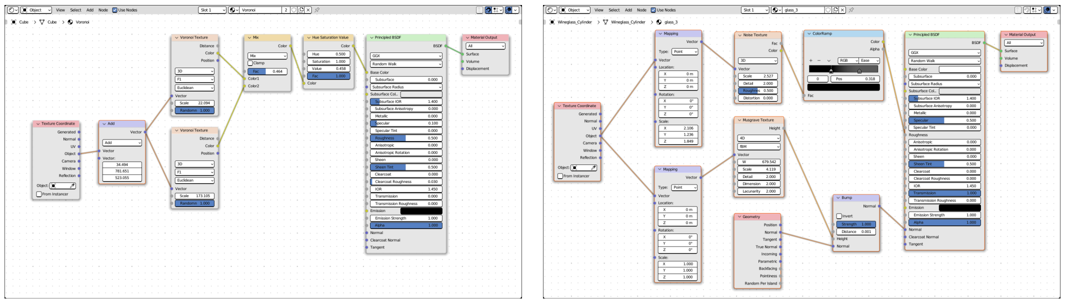

Figure 1: Random worksurace texture (left) and glass texture (right) in blender. We use textures to randomize the background and simulate imperfection in glass.

noise (from a normal distribution with $\mu = 0$, $\sigma = 0.003\,\text{m}$) and small rotation noise (from a normal distribution with $\mu = 0$, $\sigma = 0.003\,\text{rad}$) and calculate wrench resistance on 10 samples for estimating the grasp success probability. We create multi-object scenes based on single-object scenes. To do so, we sequentially sample objects on different stable poses and randomly generate a SE(2) transform that will not collide with the objects that have already been placed on the planar surface. We define the z–axis as normal and pointing outward from the worksurface. We sample x and y position from a uniform distribution between $\pm 0.2\,\text{m}$ and the z–axis rotation from a uniform distribution between $\pm\frac{\pi}{2}$. We reuse the grasps sampled for single object scenes and filter out the grasps that are in collision. We check collision between objects and grasps with the Flexible Collision Library [3]. We meshify the grasps for collision checking by using a YuMi gripper mesh model under the rigid transform given by the grasp pose.

To reflect the reachable workspace of the robot for capturing images, we record the camera intrinsics and 52 camera poses along the image capture trajectory with the physical robot. We use fewer views during dataset generation than physical capture trajectories to speed Blender rendering. For each of the simulated environments, we render images at the recorded camera poses with small translation noise ($\pm 5\,\text{mm}$) and rotation noise ($\pm 5°$). We randomize the number of lights between 1 and 5, light location, and total wattage. To speed up rendering, we reduce the numbers of rays cast to a minimum level to achieve realistic renders, and use CUDA-based renderer in Blender.

To randomize background and simulate real-world imperfections found in glass, we use two textures in Blender (Fig. 1). We use a randomized texture for the background consists of two blended random "Voronoi" nodes to produce both high and low frequency patterns. For the glass texture, we create a transparent material with an index of refraction that matches glass and many plastics. We also add random textures to simulate hazy glass and scratches. Prior work observed that NeRF performed better on real-world glass than simulated glass, observing that simulated glass had no imperfections.

We then train a NeRF model for each scene for 1000 steps, comparable to the number used on the robot in real-time, and render depth images from NeRF. We also generate ground-truth depth images using Pyrender [4] for each scene, using the same camera extrinsics as the NeRF rendered depth image. In experiments, each scene has between 1 and 3 objects and there are a total of 8667 distinct scenes, 237 held out as a test set.

C Rad-Net

C.1 Depth rendering

When rendering depth from NeRF, we use a minimum transmittance threshold of 0.9, which means that rays which have passed through a total of 0.1 density terminate. This extra sensitivity is to allow perceiving depth from transparent objects. Because density has physical meaning, in practice the value of this parameter is reusable across all scenes, in our experience not requiring tuning. During

Objects	DexNet		Rad-Net	
	Early Stop	Full Capture	Early Stop	Full Capture
Wineglass Upright	0/3	3/3	3/3	3/3
Whiskey Glass	0/3	1/3	3/3	3/3
Wineglass Sideway	0/3	2/3	2/3	2/3
Plastic Cup	0/3	0/3	3/3	3/3
Bowl	0/3	2/3	3/3	3/3
Tape Dispenser	3/3	3/3	1/3	2/3
Square Bowl	0/3	1/3	3/3	3/3
Tall Glass	0/3	3/3	3/3	3/3
Light Bulb	0/3	0/3	3/3	2/3
Average	11%	56%	89%	89%

Table 1: Detailed single object retrieval results. For each object, the experiment is repeated 3 times. The number show the success grasps out of the 3 grasps. The last row show the average success grasp over all objects. Note that all of Rad-Net's failures come from the sideways wineglass, tape dispenser, and lightbulb. The latter two suffer in performance because they are highly out of distribution shaped objects, and the former experiences a 66% success rate because grasp precision is much more important for grasping the stem or base, where a small pose error can knock the wineglass out of position.

depth rendering we ignore density which exists more than 35cm above the workspace surface, a value 2x larger than the largest test object, to help in removing floaters far above the scene.

C.2 Architecture details

The architecture we use is identical to Zhu et al. [5], except for adding an additional fully-connected layer at the output of the rotation prediction network to be more agnostic to input patch sizes. The location prediction network uses an equivariant U-Net architecture, and the rotation network is a 9-layer equivariant ResNet. The rotational equivariance operates on the cyclic group C_8 for the location prediction network and on the quotient group C_{16}/C_2 for the rotation network as top-down grasp rotation is invariant to rotations by π radians.

C.3 Training details

We use PyTorch Lightning for training, with a batch size of 64 for the rotation network and 32 for the location network. We use the Adam optimizer with learning rate 1e-3 and weight decay 1e-5. Models are trained for 100 epochs with an exponential learning rate decay of 0.994. The patches used as input to the rotation network are augmented by 5 pixels of random translation, and the location depth images are augmented by 10% translation, $\pm 5°$ shear, and a scale range of 80% to 100%.

D Additional result details

D.1 Single object

Table 1 reports the per-object success for Rad-Net and Dex-Net on early-stopped and full capture trajectories, along with a discussion of their implications in the caption.

D.2 Decluttering

Table 2 reports per-scene success for all scenes for Dex-Net and Rad-Net on Evo-NeRF as well as training NeRF from scratch, along with a discussion of the results in the caption.

D.3 Upside down glasses

In all of our experiments we test on graspable, upright objects. So, a natural question is whether Rad-Net has learned a trivial grasp function, like grasping at the edge of any round object. To answer this

Scene (N objects)	Dex-Net, NeRF updated	Rad-Net, NeRF from scratch	Rad-Net, NeRF updated
0(4)	1,4,3	4,4,4	4,4,4
1(5)	1,4,2	3,4,4	4,4,4
2(4)	1,3,4	3,4,0	0,3,3
3(4)	1,3,0	2,4,2	3,3,3
4(5)	2,3,3	4,3,2	4,2,4
5(4)	1,1,1	4,4,4	3,2,3

Table 2: Detailed decluttering results. Each scene is repeated 3 times and the method is given as many grasp attempts as the number of objects. Numbers in the parenthesis show the number of objects in this scene. Numbers in the table show the number of objects extracted after all actions finish (higher is better). Scene 2 seems to be an outlier in performance for Rad-Net, with 2 runs where no objects were cleared. This is due to a specific wineglass which Rad-Net consistently collided with during grasps, resulting in an early failure for the trial.

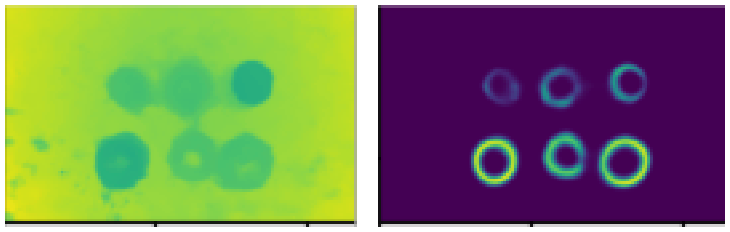

Figure 2: Heatmap output for objects including upside-down glasses. In this scene the top 3 objects are upside-down (and hence ungraspable) and the bottom 3 are graspable. The left image shows depth rendered from NeRF and the right image shows the location heatmap output by Rad-Net. Note how the heatmap activates much less on upside-down objects (<15% confidence) compared to graspable glasses (80% confidence).

question we explore what Rad-Net outputs on ungraspable, upside-down objects to sanity check its output. To do this we run a decluttering task with 3 upright and 3 upside down glasses, and inspect the confidence outputs on upside down glasses compared to upright. Given 3 actions, the system correctly removes the 3 graspable glasses and leaves the upside-down ones untouched. Fig 2 shows Rad-Net's output on this scene before the first grasp. Although the upside down glasses have ring-like heatmap outputs similar to upright cups, the highest activation on upside down glasses is 15%, which suggests that Rad-Net seems to have learned a non-trivial grasp function. Ideally, confidence on impossible grasps would be near 0, a shortcoming that could perhaps be a result of an imbalanced dataset, where more objects are upright than upside down. Cultivating a dataset with equal numbers of graspable and ungraspable poses could address this issue.

References

[1] Dawson-Haggerty et al. trimesh. URL https://trimsh.org/.

[2] J. Mahler, M. Matl, X. Liu, A. Li, D. Gealy, and K. Goldberg. Dex-net 3.0: Computing robust vacuum suction grasp targets in point clouds using a new analytic model and deep learning. In *2018 IEEE International Conference on robotics and automation (ICRA)*, pages 5620–5627. IEEE, 2018.

[3] J. Pan, S. Chitta, and D. Manocha. Fcl: A general purpose library for collision and proximity queries. In *2012 IEEE International Conference on Robotics and Automation*, pages 3859–3866. IEEE, 2012.

[4] M. Matl. Pyrender. https://github.com/mmatl/pyrender, 2019.

[5] X. Zhu, D. Wang, O. Biza, G. Su, R. Walters, and R. Platt. Sample efficient grasp learning using equivariant models. *arXiv preprint arXiv:2202.09468*, 2022.