

## Supplemental Materials

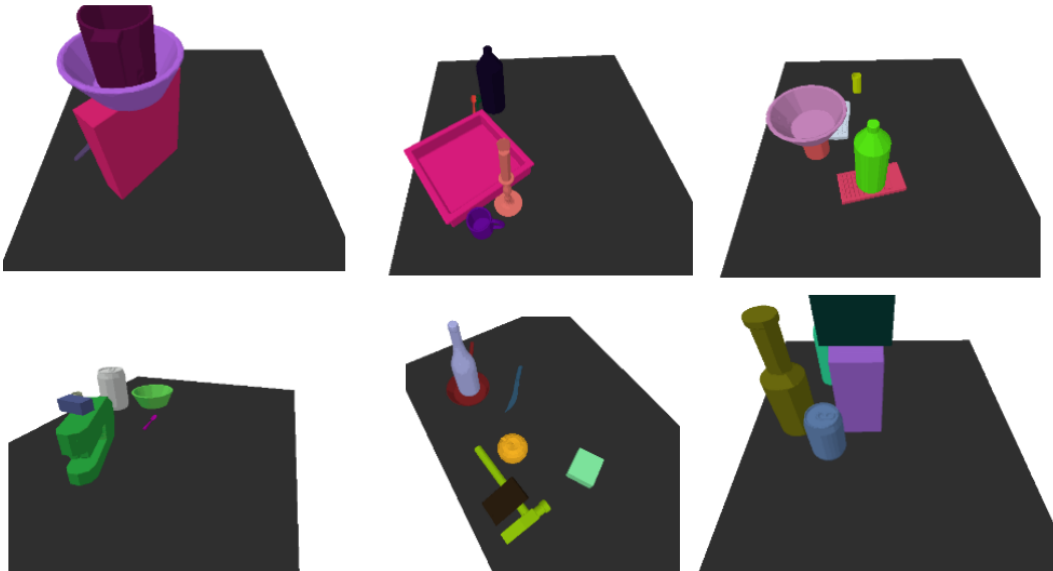
We provide two appendices. Appendix A includes a detailed explanation of our predicates, model architectures, and training. Appendix B contains additional qualitative results, including a number of real world placement examples and a discussion of our results.

### A Implementation Details

This section describes how we created our dataset and trained the model. One major advantage of our approach is that it only relies on having positive data; we automatically generate “unrealistic” data during the training process with which to train our scene-realism predictor.

#### A.1 Dataset Implementation

Each scene consists of 3 to 7 random Shapenet [1] objects in stable configurations on various surfaces, including in stacks. We include mugs, bowls, plates, bottles, pots, and various small objects, as well as boxes and cylinders of random sizes. We specifically included only objects that appear in the ACRONYM grasp dataset [2] as well as in Shapenet [1].



**Figure 1:** Examples of PyBullet scenes containing objects placed in different locations. We train on these scenes, with each object either hidden if it is query (to get scene and anchor embeddings) or hidden if it is *not* query (to get the query object embedding).

We use PyBullet<sup>1</sup> to simulate scenes. Figure 1 includes renderings of some example scenes containing the shapenet objects. Objects are moved around throughout the scene to generate data. In order to train the rotation model, we take the point cloud from before the motion, apply the correct rotation, and move it into its “observed” position. This constitutes a positive example that can be used to train our placement planner. Objects are placed either in a random orientation at a random height above the planar surface, or on top of a flat surface (which can include the table or other objects, inducing object stacks). The physics engine is then simulated forward until the objects come to rest, which results in a stable scene.

We used 5563 scenes for training our orientation model and 427 scenes for testing it. Examples of these scenes are shown in the supplemental video. In practice, we also tested a version of our model that was trained purely on static scenes, without orientation, and with only the directional predicates. This dataset contained generated 25,441 scenes with 5 images per scene from random viewpoints, resulting in a total of 221,336 relational predicates. Additional simulation results in Sec. B.2 are using this model.

---

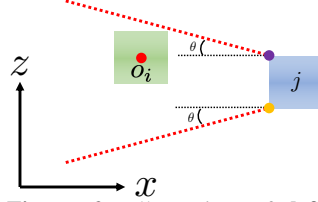
<sup>1</sup><https://pybullet.org>

We define the following predicates in our simulator: `above`, `left_of`, `right_of`, `touching`, `below`, `near`, `aligned`, `centered`, `behind`, and `in_front_of`.

### A.1.1 Directional Predicate Implementation

We define the predicates shown in Table 1. Let  $i, j$  denote the objects that will be considered for a pairwise predicate, and  $o_i$  be the 3D center of object  $i$ . We first obtain the eight bounding box corners and center for  $i, j$  in the camera frame. We use a left-handed coordinate system with the x-axis pointing right, y-axis pointing up. For the z-axis, we have it pointing towards the scene (away from the user), but we orient it such that it is parallel to the planar surface so that the above/below predicates are easy to compute.

Here, we give a high-level description of the algorithm used to determine the **left** predicate (i.e. is  $i$  to the **left** of  $j$ ?). Essentially, we compute whether  $o_i$  is within the trapezoidal volume defined by  $j$ 's bounding box corners, and an angle  $\theta$ . To implement this, we can first check this in the  $xz$ -plane. Figure 2 shows a mock-up of this in the  $xz$ -plane. Object  $i$ 's center (red dot in green object) must lie below the line defined by object  $j$ 's upper corner (purple dot) and  $\theta$ , which is denoted with the red dashed line. Similarly, it must lie above the line defined by the bottom corner (yellow dot) and  $\theta$ . Lastly,  $o_i$  must be to the left of  $j$ 's bounding box corners, which completes the trapezoidal volume definition in the  $xz$ -plane. This same computation is repeated for the  $xy$ -plane to obtain the 3D trapezoidal volume. Note that this computation assumes full knowledge of the object bounding boxes and centers, which we can obtain via the simulator.



**Figure 2:** Illustration of **left** predicate for the  $xz$ -plane.

Specifically, we consider the following set of rules:

1.  $o_i$  must be in the half-space defined by  $o_j$  upper corner and  $\theta$  in the  $xz$ -plane
2.  $o_i$  must be in the half-space defined by  $o_j$  lower corner and  $\theta$  in the  $xz$ -plane
3. do same as 1) for  $xy$ -plane
4. do same as 2) for  $xy$ -plane
5.  $i$ 's center must be to left of all  $o_j$  corners
6. All corners of object  $i$  must be to the left of  $j$ 's center

**Right** can be computed by applying the same set of rules to the flipped order of objects:  $j, i$ . Additionally, **front/behind** and **above/below** can be computed with the exact same set of rules, but considering different planes (e.g.  $xy$  and  $zy$  planes for **above/below**).

**Model-based baseline:** on point cloud data, we simply use the bounding box computed from the labeled point cloud for each object as a model-based baseline to compare against. We apply the exact same rules as in the point cloud cases.

### A.2 Other Predicates

We have four additional predicates in our dataset: `centered`, `touching`, `near`, and `aligned`. Both `touching` and `near` are defined based on mesh geometry. The mesh distance thresholds are set to 1mm for `touching` and 5 cm for `near`, but are based on the actual mesh as it appears in each scene.

The `centered` and `aligned` predicates are both based on the ground-truth pose of the objects. `centered` is true if the center of the object is within 1 mm. Two objects were considered `aligned` if the difference in orientation was less than  $\pi/20$ . In practice, our dataset was filled with Shapenet objects [1], which were not well aligned, so this proved difficult to learn; this can be fixed with better object annotations and will be explored in depth in future work.

**Model-based baseline:** We compute a reasonable model-based baseline for each approach in our dataset. For the `centered` predicate, we compute  $xy$  distance on the table and use the same threshold that appears in our dataset (1 mm). For `touching`, we compute minimum distance between point clouds and threshold it with 2.5mm. For `near`, the threshold is a 5 cm distance.

### A.3 Model Architecture

All pointnets are trained just on  $xyz$  for each point. Our encoder  $e(x)$  is a Pointnet++ network with three set abstraction layers. These are:

1. 128 points, 64 samples, scale of 0.04, and an MLP of size [3, 32, 32, 64]
2. 32 points, 32 samples, scale of 0.08, and MLP of size [64, 64, 64, 128]
3. A full set Pointnet layer with MLP of [128, 128, 128, 256]

This is followed by a single fully connected layer going from the input size to 256 with ReLU and layer norm, and then down to an  $h$  of size 128. Input size is the PointNet encoder output, concatenated with a 128-D encoding of the object center output by a second MLP.

We concatenate  $h_i$  and  $h_j$  as inputs to the predicate classifier  $p_\rho(h_i, h_j)$ . It consists of a single fully connected layer going from 256 to 128, and then to the number of predicates (nine, in our case; one was omitted because it was not applicable to real world experiments).

The prior  $\pi$  takes the concatenation of  $h_Z, h_i, h_j$ , and the predicate goal  $\vec{\rho}$ . It is then a fully connected layer from  $384 + N_{predicates}$  to 128, followed by a batchnorm. This then goes into the MDN, which predicted  $k = 5$  clusters.  $\delta$  is the 3D vector  $xyz$ .

The discriminator was a 4-layer Pointnet, with four set abstraction layers:

1. 512 points, 64 samples, scale of 0.05, and an MLP of size [3, 32, 64]
2. 256 points, 64 samples, scale of 0.10, and MLP of size [64, 64, 128]
3. 128 points, 64 samples, scale of 0.20, and MLP of size [128, 128, 256]
4. A full set Pointnet layer with MLP of [256, 256, 512]

The layer ends in a norm and a ReLU activation, a fully connected layer, and maps to a 1d output with sigmoid activation. All models were implemented using Pytorch Pointnet++ [3].

### A.4 Rotation Training

We trained a version of the model that predicts orientation instead of just  $xyz$  Cartesian transformations, as from previous work (e.g. [4]). However, training the rotation prior requires a slightly different consideration from Cartesian motions. We take two subsequent frames in our rotation placement dataset and use the frame at time  $t - 1$  for an observation at time  $t$  to extract the object’s point cloud. We compute the rotation  $\theta$  as the component of the transformation around the world’s  $z$  axis and use this rotation from  $t - 1$  to  $t$  to train the rotational component of our MDN prior.

When training models on data without rotations, we can instead use the frame at time  $t$  alone. Instead, we save extra images with and without each object rendered in the scene, to get virtual placement data.

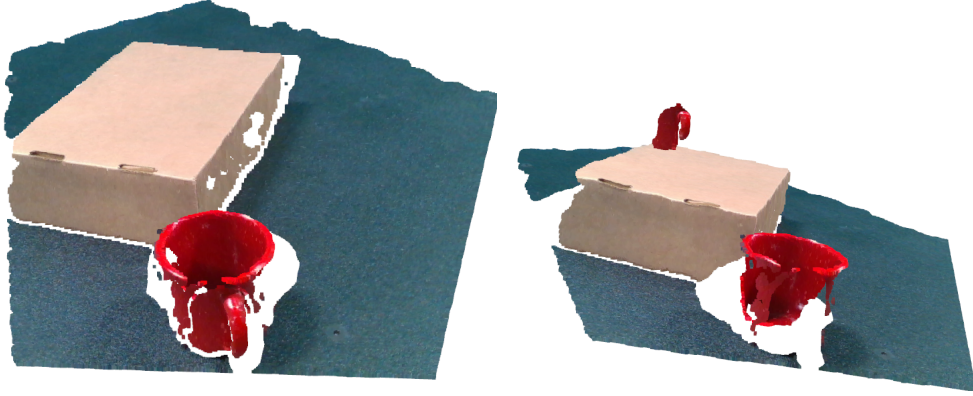
### A.5 Segmentation and Grasping

We used recent work by Xiang et al. [5] for unknown object instance segmentation, as the code is open source and available on Github<sup>2</sup>. This worked very well in many cases, but needed some minor modifications in order to be used on the real world.

In particular, we had a problem with individual pixels from around the objects being labeled as a part of those objects. This meant that even after moving an object, parts of it would be left behind, creating geometry that the planner could occasionally place things on top of. We mitigated this problem by adding a dilate and erode to each object mask, and – crucially – ignoring any 3d points that fell within the region of this dilation and erosion. This is the source of the white boundaries around all of our objects in these figures.

Even with these modifications we still saw occasional issues where segmentation problems could cause planning issues; see Fig. 3 for an example. In this case, the red mug – an object that was generally very easy to reliably grasp during our experiments – was broken up into two different pieces

<sup>2</sup><https://github.com/NVlabs/UnseenObjectClustering>



**Figure 3:** Example of segmentation causing issues when manipulating objects. In this case, the mug was broken up into two objects. Though the object was correctly placed, this made grasping very difficult because the available grasps made less sense.

before being placed “above” the large cardboard box. While the placement planning component was successful, this has clear ramifications for grasping – in particular, the generated grasps are of lower quality, and often in cases like this our motion planning will fail for safety reasons.

In the real world, grasps were computed via 6-DOF Graspnet [6], which gives us a range of grasp poses that we can use for each object. Many failures we observed in practice were due to unpredictable behavior of objects once grasped, or due to the fact that grasps would fail.

## B Additional Experiments



**Figure 4:** Real-world setup with multiple objects. The robot has an Intel Realsense D415 RGB-D camera mounted on its end effector at a known offset. We changed the scene geometry by positioning several large boxes.

We implemented our system on a real-world robot rearrangement task and show a variety of results for how our placement planner works in practice. Figure 4 shows the real-world scenario, with the

robot and a selection of the objects we tested on. We used ROS<sup>3</sup> for messaging and execution [7]. The setup also has a Microsoft Azure camera for viewing the entire scene, but this was not used in these experiments; instead, we used an Intel Realsense D415 camera to capture RGB-D images of the scene.

All tests were performed with images taken from the same camera pose to the right side of the robot, and then executed closed-loop after a manipulation plan was found. We plan motions using RRT-Connect [8].

**Object selection.** Objects were selected via a user interface, where the user would input the number associated with a particular mask (0 through  $N$ ) for each part of the query.

## B.1 Real World Placement



**Figure 5:** Placing mustard behind the bowl, with a large box in the way. There are two possible solutions to the problem: either placing the mustard on the table or placing it on top of the box. Our approach found stable placement positions in both locations.

Fig. 5 shows two examples of the model generalizing to a configuration where a large box obstructs much of the area. One major advantage of our approach is the ability to come up with a range of different placements satisfying various conditions.

We changed the scene by adding various boxes and moving things around. In one case, the system decides to place the mustard in front of the box, on the black cart surface; in the other, the mustard is placed on top of the box. Fig. 6 shows some examples of different problems solved by our method.



**Figure 6:** Examples of placement actions given individual snapshots of real-world scenes. In the top row, the robot was asked to move either the orange macaroni box to behind the milk (left), or the milk to the left of the macaroni (right), on a flat table. In the bottom row, the goal was to place macaroni (left) or juice (right) behind the milk; we added a cardboard box, which means that the object has to be placed off the table.



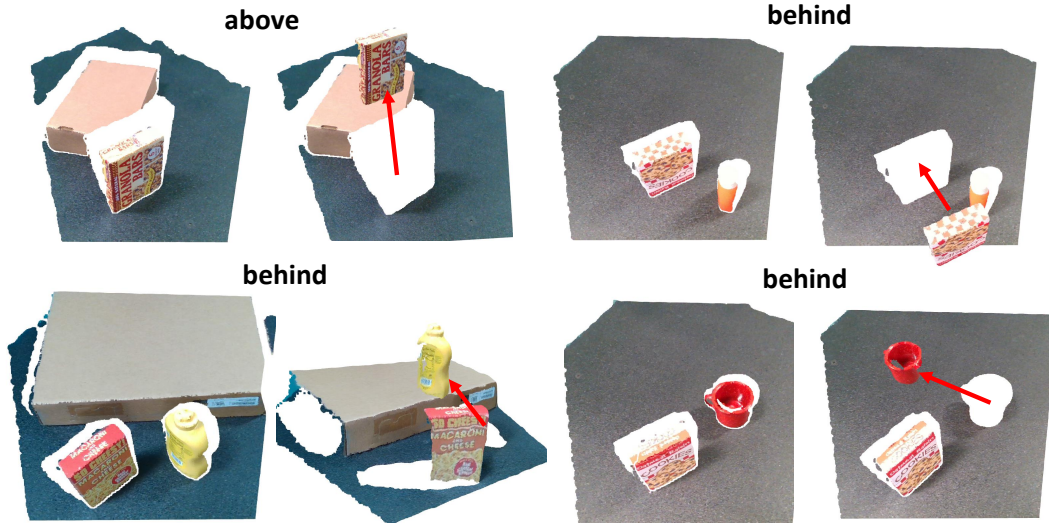
**Figure 7:** Some additional arrangement results for the “granola box” object. The planner finds a variety of poses, including stacking objects on top of one another, should that satisfy the specified predicate.

Fig. 7 shows some extra results for moving a box of granola around, placing it in different positions according to various predicate queries. Our discriminator  $f$  is very capable of finding realistic-

<sup>3</sup><https://ros.org>

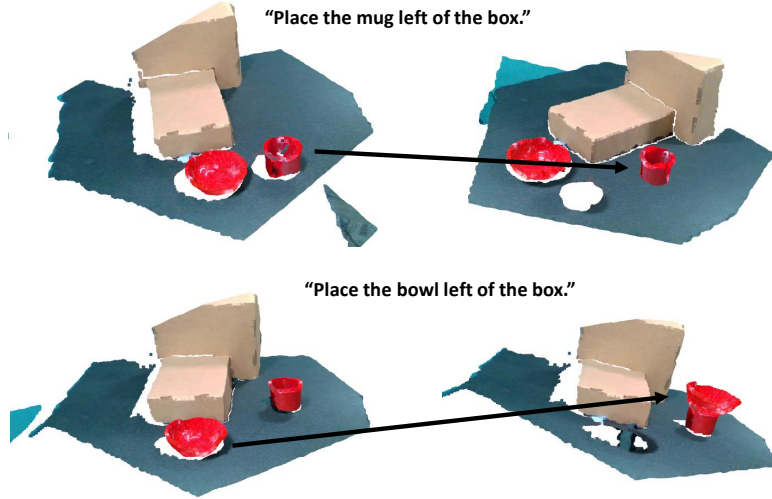


looking poses for a variety of objects; we see that it is able to plan a placement on top of the large cookie box rather easily, for example. In cases like this, execution is the main limitation preventing successes.



**Figure 8:** Examples of before and after computation of final goal positions by our planner. For each pair, on the left, we show the initial scene observation, as read in by our planning algorithm. On the right, we show the planned goal position output by the model. The white gap denotes empty space where the object was moved from.

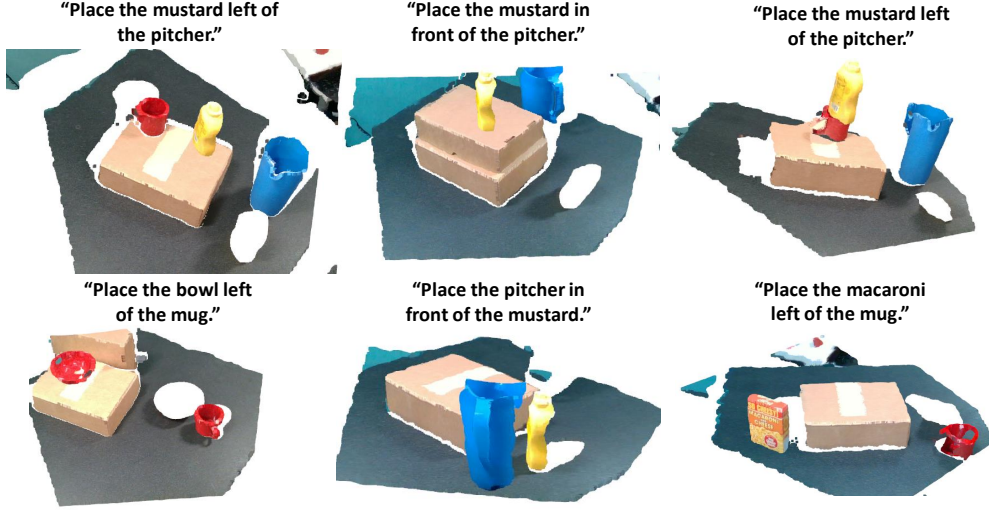
There are white gaps left in each point cloud where the granola box was moved from. These are also visible in Fig. 8, which show a number of additional successful planning queries in a very simple environment.



**Figure 9:** A sequential manipulation. The system decides to place the bowl on top of the mug in the second step, as this is a valid placement that satisfies the specified goal condition without any collisions.

Sequential manipulation results can lead to interesting outcomes. In Fig. 9, we see the results of performing a sequential manipulation, where the goal is to place the mug and bowl to the right of a large cardboard box. These sorts of actions are totally valid outcomes for our planner, and show how it has learned a variety of valid positions. Rather than picking something unrealistically close to the mug, it decides the safest thing to do is to simply stack the two.

Finally, we performed some tests in more complex or cluttered scenes, including multiple objects of which some are obstacles.

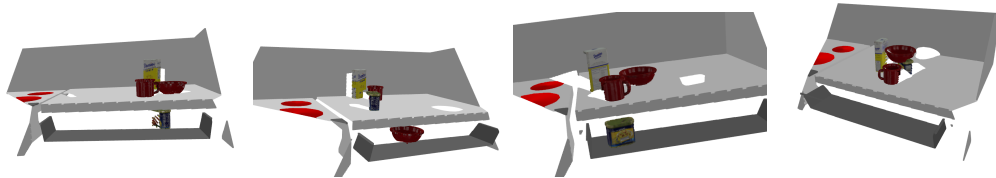


**Figure 10:** A variety of objects moved into various configurations on the table, with different heights of obstacles and different predicates chosen as commands.

Fig. 10 includes a set of experiments performed with the mustard, where the planner must adapt to the mustard bottle while dealing with a mug that might be in several different positions or with a much higher placement surface. Depending on circumstances, the planner either attempts to place the object beside or on top of the mug.

We notice here that the discriminator generally avoids placing *near* something unless explicitly told to; this is possibly a side effect of the discriminator training process. Due to differences between the simulated and real object dataset, the mustard placement in the top left of Fig. 10 is highly likely to fail, either falling into the mug or off of it.

## B.2 Cabinet Placement



**Figure 11:** Examples of interesting placements in sim. The planner is able to place objects in the drawer (if told to place below a different object) or on top of the other objects, as is appropriate.

Fig. 11 shows some additional placement results from simulation testing. These include placing the object in drawers, as appropriate, and stacking objects on top of one another, as in the lower right corner of the figure. The advantage of our model is how few assumptions it makes about the environment, which means that we can apply it to many different scenarios, even ones that we have not seen before.

Table 1 shows sensitivity (true positive rate) and specificity (true negative rate) by predicate on a set of randomly-generated simulation scenes, similar to Fig. 11. To compute classifier accuracy, we generated 100 random scenes and evaluated the classifier on each to determine if it was correct. To determine prior and planner accuracy, we sampled 100 poses for each object and determined the accuracy of each pose. The hardest predicates to classify were those based on occlusions, presumably because 3d representations are not very useful for this.

Predicate	Sensitivity (%)	Specificity (%)
Front	98.8%	89.0%
Back	96.2%	93.1%
Left	96.2%	85.0%
Right	96.5%	86.3%
Above	79.3%	97.4%
Below	79.3%	98.1%

**Table 1:** Accuracy of the predicate predictor  $p_\rho$  in randomly-generated simulated test scenes by predicate.

Predicate	Learned			Model-based			Total Examples	
	F1	Specificity	Sensitivity	F1	Specificity	Sensitivity	%True	%False
Left of	0.911	97.3%	85.6%	0.914	98.8%	85.1%	13.9%	81.6%
Right of	0.929	96.9%	89.3%	0.885	99.0%	80.0%	14.0%	86.0%
In front of	0.759	97.0%	62.3%	0.660	91.9%	51.4%	4.6%	95.4%
Behind	0.653	97.8%	49.0%	0.852	86.9%	83.4%	5.0%	95.0%
Above	0.867	99.5%	76.7%	0.784	98.4%	65.1%	4.3%	95.7%
Below	0.822	99.4%	70.0%	0.756	98.1%	61.5%	4.3%	95.7%
Near	0.869	93.7%	81.0%	0.825	98.4%	71.1%	29.0%	71.0%
Touching	0.923	97.9%	88.5%	0.418	99.2%	26.5%	12.7%	96.3%
Centered	0.659	97.6%	49.7%	0.035	100.0%	1.8%	5.6%	84.4%

**Table 2:** Accuracy of the predicate predictor  $p_\rho$  in held-out randomly-generated simulated test scenes. Some predicates in our scenes can be very difficult due to clutter and occlusions, such as *in front of*.

### B.3 Additional Predicate Comparisons

Table 2 shows extra results from our baseline comparison. This includes sensitivity and specificity, as well as prevalence in the dataset overall. We can see here that for the most part our model is better than the baseline, although not always. We should note, though, that even in the case where it is not better, it’s still an improvement since we do not need to implement some complex logic to define the predicates – as mentioned above in Sec. A.1.1, the logic for computing these arrangements isn’t trivial.

## References

- [1] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, J. Xiao, L. Yi, and F. Yu. ShapeNet: An Information-Rich 3D Model Repository. *arXiv:1512.03012*, 2015.
- [2] C. Eppner, A. Mousavian, and D. Fox. ACRONYM: A large-scale grasp dataset based on simulation. In *ICRA 2021*, 2020.
- [3] E. Wijmans. Pointnet++ pytorch. [https://github.com/erikwijmans/Pointnet2\\_PyTorch](https://github.com/erikwijmans/Pointnet2_PyTorch), 2018.
- [4] A. H. Qureshi, A. Mousavian, C. Paxton, M. C. Yip, and D. Fox. Nerf: Neural rearrangement planning for unknown objects. *arXiv preprint arXiv:2106.01352*, 2021.
- [5] Y. Xiang, C. Xie, A. Mousavian, and D. Fox. Learning rgb-d feature embeddings for unseen object instance segmentation. In *Conf. on Robot Learning*, 2020.
- [6] A. Mousavian, C. Eppner, and D. Fox. 6-dof grasping: Variational grasp generation for object manipulation. In *Intl. Conf. on Computer Vision*, 2019.
- [7] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, 2009.
- [8] J. J. Kuffner and S. M. LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Intl. Conf. on Robotics and Automation*, 2000.