

VIII. APPENDIX

A. Data Generation

Point completion. We consider 100 random viewpoints for each object from the YCB dataset from which we obtain the partial point clouds inputs. A sample of the random views on three YCB objects is shown in Figure 6.



Fig. 6: *Point completion Data Generation:* Given the mesh/point cloud of the object, we render depth maps from 100 random views around a sphere of 1 unit looking at $(0, 0, 0)$, with the object in centered at the origin.

Dynamics modeling. For all tasks across ManiSkill and DexMimicGen, we write a custom wrapper to obtain the complete point clouds from the simulation itself by sampling points from objects/parts/geom meshes. Further, to handle point-to-point correspondences, we only sampled the points at the canonical frame and applied rigid body pose transformations for all the desired entities in the scene. This significantly boosted the training data collection time as well as reduced the need to store point clouds at each step (only storing one canonical point cloud and $\text{len}(\text{traj})$ transformation matrices).

B. Model Details & Hyperparameters

3DPWM dynamics modeling. For training the dynamics model, we adopt the PTV3 architecture [51], replacing batch normalization with layer normalization following Sonata [62]. To better suit the smaller dataset size, we reduce the depth of each of the four PTV3 encoding stages from the default configuration to $[2, 2, 2, 2]$. Relative positional encoding (RPE) helps to capture spatial relationships between the points, but enabling it throughout the network makes it hard to use FlashAttention [63], [64] to speed up the training process. Instead, we enable RPE in alternating encoder and decoder blocks. Accordingly, we adjust the patch size for neighborhood computation from 256 in the layers where RPE is disabled to 48 when RPE is enabled. The bottleneck per-point feature dimension is set to $d = 256$, while the number of input points M depends on the grid resolution and scene complexity, typically ranging from 200 to 400 points. For action conditioning, both the `CrossAttn` and `SelfAttn` modules consist of three layers with eight attention heads each. The decoder outputs 128-dimensional per-point features, which are passed through a four-layer MLP to predict per-point velocities. Wherever applicable, we employ FlashAttention [63], [64] across all attention modules in PTV3, including the cross- and self-attention layers, to improve training efficiency. We provide

Config/Hyperparameter	Value
Point Completion	
jitter prob	0.3
jitter mean	0.01
jitter variance	0.03
global feature channels	512
encoder channels	[32, 64, 128, 256]
decoder upsample factors	[2, 2, 2]
decoder hidden channels	64
seed generator blocks	4
seed generator - attn heads	8
seed generator - attn dims	384
seed generator - upsample factor	2
seed generator - output dim	256
Point Dynamics model	
serialization pattern	Z + TZ + H + TH
patch interaction	Shift Order + Shuffle Order
positional encoding	[CPE, RPE, CPE, RPE]
stride	[2, 2]
encoder depth	[2, 2, 2]
encoder channels	[32, 128, 256]
encoder num heads	[8, 8, 8]
encoder patch size	[256, 48, 256]
decoder depth	[2, 2]
decoder channels	[128, 256]
decoder num heads	[8, 8]
decoder patch size	[256, 48]
mlp ratio	4
qkv bias	True
drop path	0.3
pre norm	True
shuffle orders	True
MPPI Planner	
planning horizon	20-30 steps
# num samples	200
# top candidates	64
lambda_	0.7
noise σ	5
Training Hyperparameters	
loss	HuberLoss, delta=0.25
optimizer	AdamW
learning rate	0.0001
scheduler	StepLR, step_size=30, $\gamma = 0.1$

TABLE III: Hyperparameter values for different modules in our method.

detailed hyperparameters for the completion module, dynamics training as well as for planning in Appendix VIII-B. We outline the hyperparameters for all the module of 3DPWM in Table III.

SAM3 Prompting: Given the task description, we first prompt Gemini Robotics 1.5 VLM [48] to identify task-relevant objects in the scene by outputting their 2D point

locations. We then use these points as prompts to SAM3 [47] to segment the corresponding objects.

We use the following prompt to instruct the vision-language model to identify task-relevant objects and provide 2D point prompts for segmentation:

You are a vision-language model assisting a robot with a manipulation task.

Given:

1. A natural-language task description.
2. An RGB image of the current scene.

Your goal is to identify all objects that are relevant to successfully completing the task and provide one or more 2D point locations for each relevant object.

Instructions:

1. Read the task description carefully and determine which physical objects the robot must interact with, manipulate, move, avoid, or otherwise reason about to complete the task.
2. In the image, locate each task-relevant object.
3. For each relevant object, select a 2D point that lies clearly INSIDE the visible region of that object.
 - The point should be near the center of the visible portion of the object whenever possible.
 - Do not place the point on the object's boundary.
 - Do not place the point on the background, another object, or an occluding object.
 - Prefer points that are visually unambiguous and far from object boundaries.
 - If an object is partially occluded, place the point on the visible portion of the object.

...

Output format:

```
[
  {
    "object": "<short object description>",
    "point": [x, y]
  },
  ...
]
```

Example:

Task: "Pick up the red mug and place it

inside the bowl."

Output:

```
[
  {
    "object": "red mug",
    "point": [312, 241]
  },
  {
    "object": "bowl",
    "point": [487, 326]
  }
]
```

Be precise: every point must fall inside the corresponding object's visible pixels in the image.

C. Cost Function Specification

One of the key goals of learning a point based world model in the observation space, is to be able to specify cost functions to enable planning. This requires that the rollouts produced by the dynamics model are of high fidelity – so that the cost/reward computation on the rolled out point clouds is accurate. Since we perform point completion, which enables high fidelity long-horizon rollouts, we can use cost functions which can be defined on point clouds directly. We prompt Gemini 2.5-Flash [60] with the following prompt to obtain cost functions that can be defined in the point cloud observation space.

You are a vision-language model assisting a robot with a manipulation task.

Your goal is to generate a task-specific cost function for evaluating predicted robot states during trajectory planning.

You are given:

1. A natural-language task description:

<TASK>

2. A JSON file containing objects identified as potentially relevant to the task:
<OBJECTS_JSON>

The objects JSON contains the objects detected in the scene and their 2D image point locations. It may contain:

- * Objects that must be manipulated to complete the task.
- * Objects that serve as destinations or targets.

- * Objects that provide contextual information needed to complete the task.
- * Objects that are not explicitly mentioned in the task but may be relevant because the robot could collide with them or they may constrain the robot's motion.

The generated cost function will be evaluated on 3D point clouds predicted by a robot world model. The point clouds represent the predicted geometry of objects and robot components at future states along candidate trajectories.

Cost function interface

Generate a Python/PyTorch function with the following interface:

```
```python
def cost_fn(cfg, point, info):
 """
 Task-specific cost function.

 Args:
 cfg: Configuration object. In particular, cfg.device specifies the PyTorch device on which tensors should be created.

 point: Dictionary containing predicted 3D point clouds for objects and robot components. Keys correspond to object names from the provided objects JSON and may also include robot components such as 'left_gripper' and 'right_gripper'.

 Each point cloud is a PyTorch tensor containing 3D coordinates.

 info: Dictionary containing additional task/environment information that may be required to evaluate the task, such as goal positions, grasp/contact state, or task stage. Additional information is available under info['extra'] when provided.

 Returns:
 A single scalar PyTorch cost. Lower cost indicates a more desirable state.
 """
 ...
 return total_cost
```
```

Requirements

1. **Infer the task objective**

Carefully interpret '<TASK>' and determine what the robot must accomplish, including any intermediate stages or object relationships required for successful completion.

2. **Use the objects JSON**

Use the provided objects to determine which objects should influence the cost.

Do not invent task objects that are not present in the JSON.

3. **Include potentially relevant objects**

Do NOT include clearly irrelevant scene elements such as the background or image artifacts. However, include objects that could potentially be relevant to execution, even if they are not explicitly mentioned in the task.

In particular, consider objects that the robot arm, gripper, or manipulated object could collide with or that could constrain the robot's motion.

4. **Operate on predicted point clouds**

The cost function should primarily operate on the 3D point clouds in 'point'.

For example, object geometry may be summarized using:

- * Point-cloud centroids.
- * Distances between point clouds.
- * Distances between object centers.
- * End-effector-to-object distances.
- * Object-to-target distances.
- * Relative positions.
- * Spatial overlap or containment.
- * Proximity to potential obstacles.

Choose geometric quantities appropriate for the specific task.

5. **Use 'info' when necessary**

Use 'info' for information that cannot be obtained reliably from the point clouds, such as:

- * Whether an object is currently grasped.
- * Task stage.
- * Goal positions.

* Other task-specific environment state.

Do not assume privileged information is available unless it is provided through 'point' or 'info'.

6. ****Measure task progress****

The cost should provide a meaningful measure of progress toward task completion.

For multi-stage tasks, the cost may change depending on the current task stage. For example, it may first encourage reaching an object and then encourage moving the object toward its destination after grasping.

7. ****Collision avoidance****

Include collision or near-collision costs when potentially relevant objects could interfere with the robot's execution.

The collision cost should increase as the robot or manipulated object approaches a potential obstacle.

Collision avoidance should complement, rather than overwhelm, the primary task objective.

8. ****Smoothness****

Prefer smooth cost functions that provide useful ranking information between candidate states, including when none of the candidates completely succeeds.

Prefer continuous functions such as Euclidean distances, 'torch.tanh', exponentials, or other smooth distance-based penalties over hard binary thresholds when appropriate.

9. ****Scalar output****

The function must return a single scalar PyTorch tensor:

```
```python
return total_cost
```
```

Lower cost must correspond to a better state.

10. ****Task-specific design****

Do not produce a generic cost function. Construct the cost specifically from the provided task description and

objects JSON.

11. ****Keep it simple****

Use only the terms necessary to capture the essential task objective and relevant collision constraints. Avoid unnecessarily complicated heuristics.

12. ****PyTorch compatibility****

Use PyTorch operations for geometric calculations so that the function can operate directly on the tensors in 'point'.

Output format

Return:

1. A brief description of the task objective and the relevant object roles.

2. A brief description of the cost terms and what each term measures.

3. The complete Python/PyTorch implementation:

```
```python
def cost_fn(cfg, point, info):
 ...
 return total_cost
```
```

4. A brief list of assumptions about the available keys in 'point' and 'info'.

The final implementation must be directly usable as a task-specific cost function for ranking predicted robot trajectories.

Alternatively, manually constructed cost functions can also be used. We show an example of the cost function adapted in a very straightforward manner from ManiSkill [55] in a pytorch-like code 1.

```
1 def cost_fn_pick_cube(cfg, point, info):
2     """
3     Cost function for the PickCube-v1 task.
4     """
5
6     goal_xyz =
7         torch.tensor(info['extra']['goal_pos'],
8                       device=cfg.device)
9
10    left_gripper_xyz = point['left_gripper']
11    right_gripper_xyz = point['right_gripper']
12    cube_xyz = point['cube']
13
14    cube_center_xyz = cube_xyz.mean(dim=0)
15    tcp_xyz = (left_gripper_xyz.mean(dim=0) +
16              right_gripper_xyz.mean(dim=0)) / 2
```

```

16 tcp_to_cube_dist = torch.norm(cube_center_xyz -
17 tcp_xyz, dim=0)
18 reaching_reward = 1.0 - torch.tanh(5.0 *
19 tcp_to_cube_dist)
20 cube_to_goal_dist = torch.norm(cube_center_xyz
21 - goal_xyz, dim=0)
22 place_reward = 1.0 - torch.tanh(5.0 *
23 cube_to_goal_dist)
24 contact_reward = 1.0 if info['grasped'] else 0.0
25 if info['grasped']:
26     total_cost = -place_reward
27 else:
28     total_cost = -(reaching_reward +
29 contact_reward)
30 return total_cost

```

Listing 1: Cost function for the PickCube task that operates on **3DPWM**’s rolled out point clouds.

Since the notion of contacts is not baked in explicitly, we perform grasping when the gripper is within 2.5 cm of the object (i.e distance between `tcp_xyz` and `cube_center_xyz` is ≤ 0.025) and assume a success (i.e `info['grasped'] = True`). Alternatively, we can also try computing the intersection between the left-gripper, object and the right-gripper to determine a contact, however, in practice we found it hard to tune the threshold for detecting the contact. We leave addressing this issue in point world models as a direction for future work.

D. Failure Scenarios

Point completion. First we discuss some of the failures of point cloud completion module on real world objects. We show a sample of completion success as well as failures in Figure 7. The failure scenarios primarily arise due to a mismatch between the training partial point cloud distribution and the real-world partial point cloud distribution. As mentioned in Sec. VI, our current data generation pipeline for point cloud completion does not include occlusions cause by other objects in the scene. This is because we generate partial point clouds of the objects by considering individual meshes of the objects and generating depth maps from 100 random viewpoints and backprojecting the depth to obtain the point cloud. This process, by design, does not include various kinds of occlusions caused due to the presence of other objects. Further, the noisy generation of points outside the boundary as seen in Fig 7 is also a common failure point in the pipeline.

E. Additional Experiments on **3DPWM** robustness towards lighting conditions and camera viewpoints

We also experimented with 2 additional camera viewpoints as well as changing the lighting conditions to test the robustness of **3DPWM**. It has been shown that policies trained with calibrated point clouds are generally more robust to such lighting and camera viewpoint changes [20], [65]. This was further corroborated by our results on PickApple shown in Figure 8. Specifically, we observe that **3DPWM** maintains its higher performance when compared with ParticleFormer⁺.

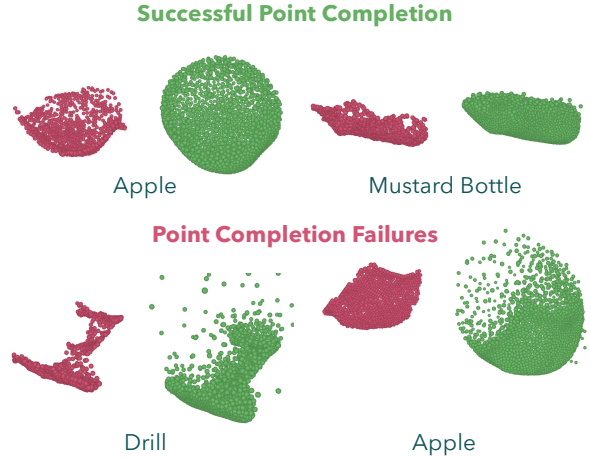


Fig. 7: *Point completion success & failures*: We show few samples of the fine-tuned point cloud completion model. (Top) Successful point cloud completions on partial point clouds of apple and mustard bottle. (Bottom) Failure scenarios of the completion on Drill and Apple from the YCB dataset.

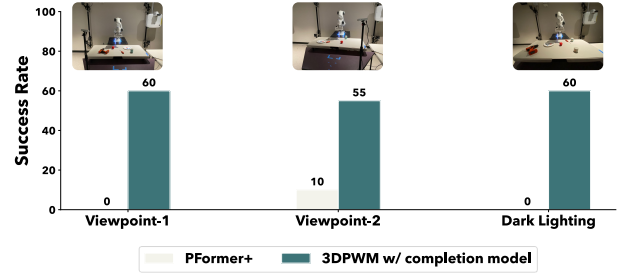


Fig. 8: *Lighting & Viewpoint condition*: We tested the robustness of **3DPWM** on PickApple task with two additional viewpoints as well as darker lighting condition. (Each plot is averaged over 20 trials.)

F. Additional Experiments: Comparison to GNN based methods

TABLE IV: Comparison between 3DPWM and graph neural network GBND

| Task | Chamfer L1 ($\downarrow$) | | Reward Corr (r) ($\uparrow$) | |
|-------------|-----------------------------|-------|------------------------------------|-------|
| | GBND [24] | 3DPWM | GBND [24] | 3DPWM |
| PickCube | 0.185 | 0.009 | 0.37 | 0.86 |
| Pick YCB | 0.21 | 0.028 | 0.33 | 0.85 |
| Box Cleanup | 0.301 | 0.022 | 0.16 | 0.83 |

Gaussian Splatting based methods like [24] comprise of (a) Gaussian point-based dynamics and (b) splatting-based rendering. Since **3DPWM** focuses on modeling the dynamics and *not* the rendering, we compare against the GNN-based dynamics model from [24] in Table IV. As shown in [19] and confirmed by our results – GNN-based dynamics underperforms transformer-based approaches. Integrating a rendering pipeline into **3DPWM** remains a future work that we wish to address with **3DPWM** as the dynamics model.

TABLE V: Chamfer distance and reward correlation on both variants of ParticleFormer (+ and *) (**Best** and **second best**).

| Task | Avg. rollout len. | Chamfer L1 ($\downarrow$) | | | Reward Corr (r) ($\uparrow$) | | |
|-------------|-------------------|-----------------------------|----------------------|--------------|------------------------------------|----------------------|--------------|
| | | PFormer ⁺ | PFormer [*] | 3DPWM | PFormer ⁺ | PFormer [*] | 3DPWM |
| PickCube | 66 | 0.080 | 0.132 | 0.009 | 0.61 | 0.41 | 0.86 |
| Box Cleanup | 218 | 0.132 | 0.225 | 0.022 | 0.30 | 0.19 | 0.83 |

G. Additional Experiments: Sim2Real in Cluttered Environment

We further evaluate the planning capabilities of **3DPWM** in cluttered environments across two tasks: PickPlaceApple and StackCube. In PickPlaceApple, the apple is initialized near a group of cluttered objects, and the cost function incorporates a heuristic penalty for collisions with surrounding objects. StackCube requires stacking a yellow cube on top of a blue cube in a moderately cluttered environment. This task is particularly challenging, as precise placement is critical; notably, all observed failures on StackCube are near-misses, as illustrated in Figure 9 (right).



Fig. 9: *Sim2Real on Cluttered Environments*: (Left) We evaluate **3DPWM** on two cluttered-environment tasks: PickPlaceApple, in which the robot must pick an apple surrounded by clutter, and StackCube, in which the robot must stack cubes amid a cluttered scene. (Results are averaged over 20 trials per task.) (Right) Failure scenarios of **3DPWM** on StackCube occurs at the very end during placement of the yellow cube on top of the blue cube.

H. Baselines: Particleformer [19]

Here we lay out the specific details of our re-implementation of ParticleFormer [19]. Each partial point input has point velocities and one-hot vector representing instance segmentation. We run two architecture variants (i) ParticleFormer^{*}, which is re-implementation of the the ParticleFormer baseline with 3 transformer layers as in the original paper, and (ii) ParticleFormer⁺ an equivalent variant of encoder and decoder that uses our PTV3 architecture [51]. Unlike **3DPWM**, we train both these variants on segmented partial point clouds and supervise the dynamics model with the hybrid loss (a combination of Chamfer and Hausdorff losses) as proposed by the authors.

We report the performance of ParticleFormer^{*} here in Table V along with the copied results of ParticleFormer⁺ from Table I. We find that the original ParticleFormer with 3-layers of transformer is insufficient to capture the dynamics of both PickCube as well as Box Cleanup tasks and fares lower to ParticleFormer⁺.

I. Environments

We evaluate our dynamics model design across 6 different tasks – PickCube, StackCube and PickYCB from the Maniskill simulator [55], MugCleanup from MimicGen [56] and TwoArmBoxCleanup and TwoArmDrawerCleanup from Dexmimicgen [57]. A sample snapshot of the environment is shown in Figure 10.



Fig. 10: *Environments used to evaluate 3DPWM*:

PickCube: Lift the cube from the table to the goal position in the air specified by a 3D point in space.

StackCube: Stack the red cube on top of green cube.

PickYCB: Pick up the YCB object from the table to the goal position in the air specified by a 3D point in space.

MugCleanup: Pick and place the mug in side the drawer and close it.

TwoArmBoxCleanup: Pick up the lid of the box using two dexterous hands and place it on the box.

TwoArmDrawerCleanup: Pick and place the mug using two dexterous hands in side the drawer and close it.

For all the tasks with parallel jaw gripper, we use the end-effector delta actions, and for dexterous tasks we use 24-dim end effector actions (6-dim for pose, and 6 for finger control on each arm).

| Task | Observation Points | Action dim |
|---------------------|--------------------|------------|
| PickCube | 1200 | 7 |
| StackCube | 1500 | 7 |
| PickYCB | 1200 | 7 |
| MugCleanUp | 8192 | 7 |
| TwoArmBoxCleanup | 10000 | 24 |
| TwoArmDrawerCleanup | 12000 | 24 |

TABLE VI: Number of points for complete point cloud and action spaces of each of the tasks.

J. Additional experiments we tried that did not work.

In this section we lay out some of the additional experiments we tried but did not see success and believe that this would be helpful to share with the community to improve upon.

Using SAM3D instead of custom trained point completion. We experimented with SAM3D [26] but found that

the completions, while generalizable to various commonly found objects *do not align* with the partial input despite providing sensor depth from the camera (instead of using DepthAnything or other foundation depth models). This is a known issue (see Issues [#79](#) and [#162](#) on the SAM3D github repository.)

Using Depth Foundation models to avoid noisy completions. We also tested using MOGE-2 [\[66\]](#) and DepthAnything3 [\[67\]](#) since they have shown impressive performance for depth estimation. However, we observed that while the depth was very clean and so were the resulting backprojected point clouds – these models still suffered with two major challenges which made decision making difficult: (a) similar to SAM3D, these depth foundation models suffered with object misalignment in the real world, and (b) they tended to miss out smaller details (depending on the viewpoint of the object) of the objects that could potentially be crucial for manipulation.