# Supplementary Materials

Yixuan Wang[1,2], Leonor Fermoselle[2], Tarik Kelestemur[2], Jiuguang Wang[2], Yunzhu Li[1]

## I. Method

### A. IoU Computation

To compute the IoU of two point clouds, we first compute the distance between each point from two point clouds. If a point's minimum distance to another point cloud is smaller than the predefined threshold $\tau$, this point is considered an "intersection". After counting all intersections, we can compute IoU by dividing the intersections by the total point number. The detailed algorithm is shown in Algorithm 1.

**Algorithm 1** IoU Between Point Clouds

```
Input: Two point clouds $A \in \mathbb{R}^{N\times 3}$, $B \in \mathbb{R}^{M\times 3}$; threshold $\tau$
Output: IoU matrix IoU
1: if $|A| = 0$ or $|B| = 0$ then return 0
2: Compute distances $D$ between all points in $A$ and $B$:
       $D_{k,l} = \|A[k] - B[l]\|, \quad \forall k = 1,\dots,N; \quad l = 1,\dots,M$
3: Compute minimum distances for each point:
       $d_k^{(A)} = \min_l D_{k,l}, \quad \forall k = 1,\dots,N$
       $d_l^{(B)} = \min_k D_{k,l}, \quad \forall l = 1,\dots,M$
4: Create masks based on threshold $\tau$:
       $\text{mask}_A[k] = \begin{cases} 1, & \text{if } d_k^{(A)} < \tau \\ 0, & \text{otherwise} \end{cases}$
       $\text{mask}_B[l] = \begin{cases} 1, & \text{if } d_l^{(B)} < \tau \\ 0, & \text{otherwise} \end{cases}$
5: Compute sums of masks:
       $S_A = \sum_{k=1}^{N} \text{mask}_A[k], \quad S_B = \sum_{l=1}^{M} \text{mask}_B[l]$
6: Compute IoU:
       $\text{IoU} = \frac{S_A + S_B}{N + M + \varepsilon}$
7: return IoU
```

### B. Unknown Space

Using depth image and camera parameters, we could create a voxel representation of the space. For each voxel, we can have labels, such as `unexplored`, `free`, `unknown`, and `outside`. The definition of each label is listed below:

- `unexplored`: If a voxel is never viewed by a sequence of camera observations, the voxel is labeled as `unexplored`.
- `free`: If a voxel is viewed by a sequence of camera observations and it is in the free space, the voxel is labeled as `free`.
- `unknown`: If a voxel is viewed by a sequence of camera observations and it is occluded, the voxel is labeled as `unknown`.
- `outside`: If a voxel is viewed by a sequence of camera observations and it is outside of the room, the voxel is labeled as `outside`.

Such a voxel representation is critical for VLM decision. For example, when there are a lot `unknown` voxels inside the cabinet or behind the box, VLM will utilize such information regarding the unknown space to decide where to interact.

### C. Relation Detection

Given such voxel representation, we can define the following object relations:

- `of`: If a child object can be possibly a part of the parent object, such as a handle is possibly a part of a cabinet, and their point cloud centroids are close enough, the child object node is `of` the parent object node.
- `inside`: If a child object is found after opening or flipping a parent object, the child object is `inside` the parent object node.
- `on`: If a child object's bounding box is within the parent object's bounding box in the x-y plane, and the child object's lowest z value is close to the parent object's highest z value, the child object is `on` the parent object node.
- `under`: If a child object is found after sitting down or lifting, the child object is `under` the parent object node.
- `behind`: If a child object is found after pushing a parent object aside, the child object node is `behind` the parent object node.

### D. Graph Serialization

We serialize the graph using the depth-first search. The detailed algorithm is presented in Algorithm 2.

https://bdaiinstitute.github.io/curiousbot/
[1]Columbia University [2]Boston Dynamics AI Institute

**Algorithm 2** Serialize Graph

**Input:** A graph $G$ with nodes $V$ and edges $E$

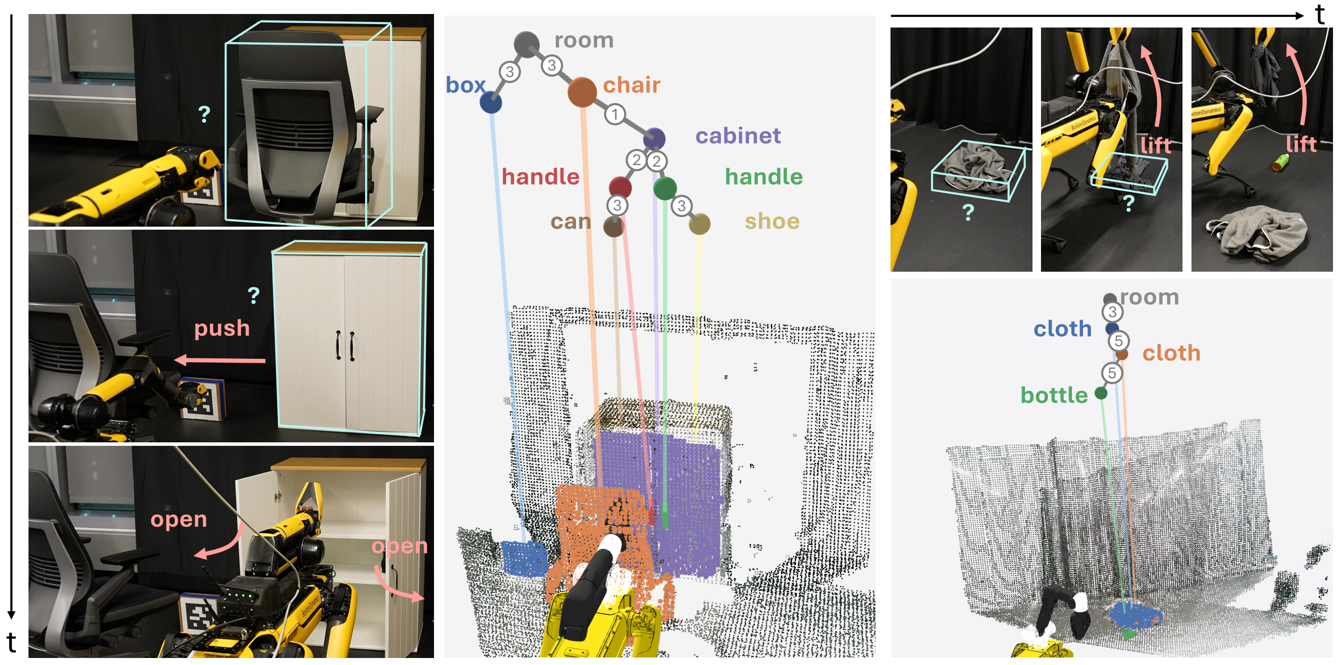
**Output:** A string `texts` representing the serialized graph

```
1:  Initialize texts ← "root\n"
2:  Initialize to_visit ← empty stack
3:  Add root_node to to_visit
4:  while to_visit is not empty do
5:      // Append the text of the current node to final texts
6:      Pop first node curr_node from to_visit
7:      Obtain text curr_text of curr_node
8:      Append curr_text to return_texts
9:
10:     // Find children nodes and add to to_visit
11:     child_nodes ← curr_node.get_children()
12:     for all child_node in child_nodes do
13:         Add child_node to to_visit
14: return return_texts
```

## E. Prompts Examples

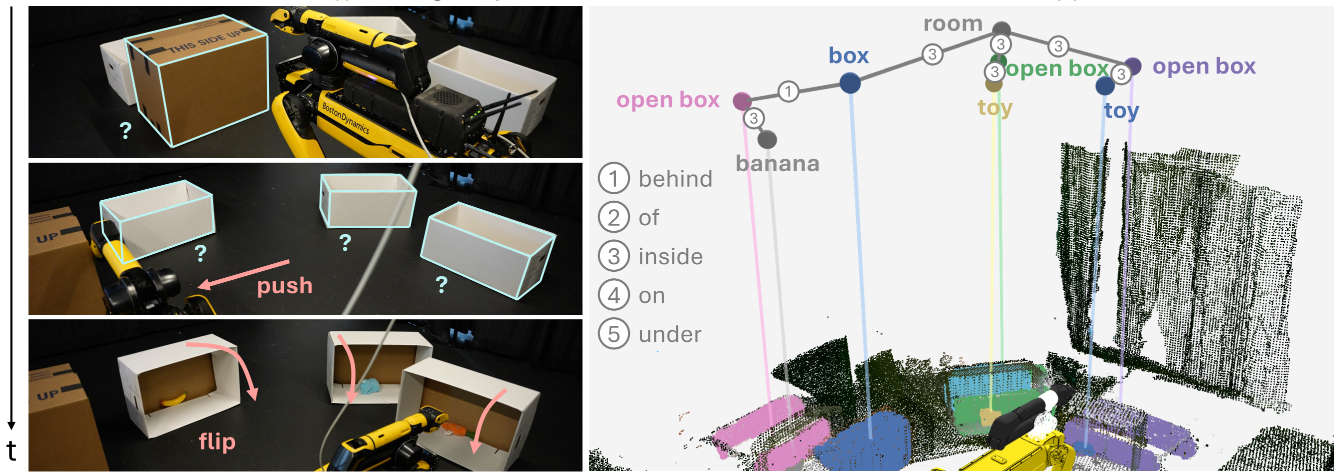
Here are all the prompt examples we use. We provide minimal prompt examples to the robot for task planning.

```
# explore one cabinet with one handle
graph:
root
\---cabinet_0
    \---handle_1 [obstruction]

# answer
action:
open handle_1 # affected_objects:
cabinet_0, handle_1

# explore one cabinet with multiple
handles
graph:
root
\---cabinet_0
    |---handle_2 [obstruction]
    \---handle_1 [obstruction]

# answer
action:
open handle_2 # affected_objects:
cabinet_0, handle_2
open handle_1 # affected_objects:
cabinet_0, handle_1

# explore one cabinet with multiple
handles, while some handles are opened
graph:
root
|---chair_3 [obstruction]
|---box_4 [obstruction]
|---cloth_5 [obstruction]
\---cabinet_0
    |---handle_1 (opened)
    \---handle_2 [obstruction]

# answer
action:
push chair_3 # affected_objects: chair_3
push box_4 # affected_objects: box_4
lift cloth_5 # affected_objects: cloth_5
open handle_2 # affected_objects:
cabinet_0, handle_2

# explore several cabinets and
obstruction objects
graph:
root
|---cabinet_0
|   |---handle_1 (opened)
|   |   \---object_3
|   \---handle_2 (opened)
|       \---object_5
|---object_7
|---box_9 [obstruction]
\---cabinet_6
    \---handle_4 [obstruction]

# answer
action:
open handle_4 # affected_objects:
cabinet_6, handle_4
push box_9 # affected_objects: box_9

# explore several cabinets and
obstruction objects
graph:
root
|---cabinet_0
|   |---handle_1 (opened)
|   |   \---object_3 (moved)
|   |       \---object_8
|   \---handle_2 (opened)
|       \---object_5 (moved)
|---object_7
|---box_9
\---cabinet_6
    \---handle_4 (opened)

# answer
action:
none

# explore several cabinets and
obstruction objects
graph:
root
|---cabinet_0
|   |---handle_1 (opened)
|   |   \---object_3 (moved)
```

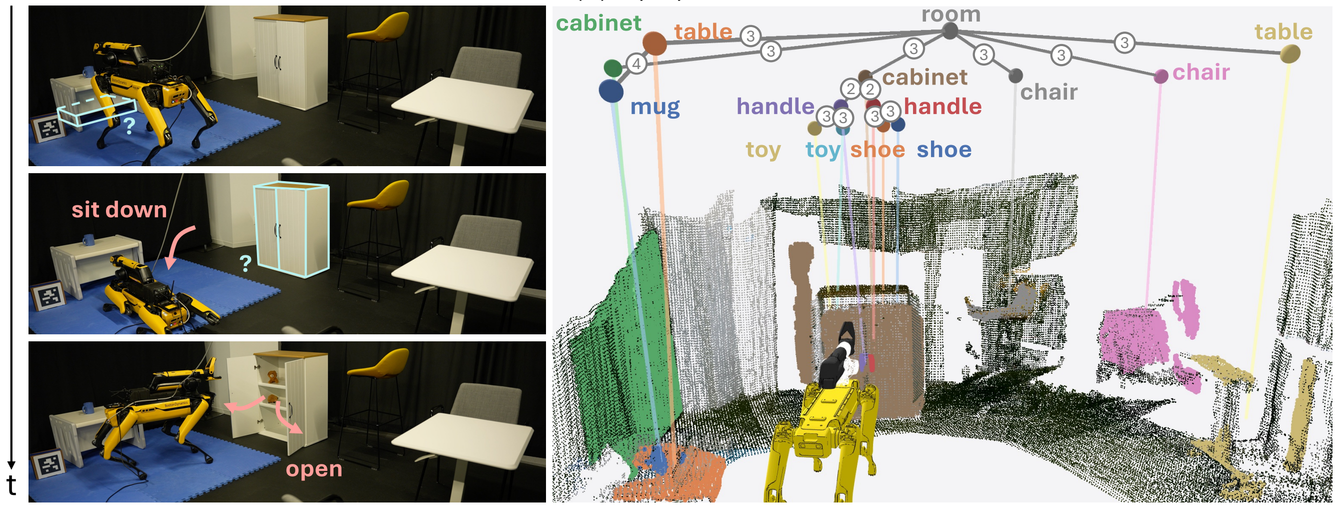


(i) Pushing Away Chair

(ii) Lift Cloth



(iii) Flip Open Boxes



(iv) Explore Household Scene

Fig. 1: **Qualitative Results.** We evaluate our system's exploration capabilities across various tasks, including pushing the chair aside to reveal space behind it, lifting cloth to check underneath, flipping open boxes to inspect the contents, and exploring a household scene. These tasks showcase the system's ability to generalize across different object types, scenarios, and object relations. Additional tasks can be found on the project page.

```
|   |       \---object_9 (moved)
|   \---handle_2 (opened)
|       \---object_5 (moved)
|---object_7
\---cabinet_6
    \---handle_4 (opened)

# answer
action:
none

# explore several cabinets and
obstruction objects
graph:
root
|---sealed_box_1
|---sealed_box_2
\---open_box_3 [obstruction]

# answer
action:
flip open_box_3 # affected_objects:
open_box_3
```

## II. Experiments

### A. Full Qualitative Results

Due to the page limitation, we only show partial results in our main paper. Here we list more complete qualitative results in Figure 1.