Supplementary Materials for GENMANIP

The supplementary materials are organized as follows:

1. **Video demo:** We provide a supplementary video demonstration of GENMANIP, which includes additional examples of task scenario visualization and the real-world deployment of modular systems.
2. **Experimental setups** are shown in Sec. 1.
3. **GENMANIP task and scenario generation**:
   (a) Prompts for task-oriented scene graph generation are included in Sec. 2.
   (b) Pseudo code for layout generation given a scene graph is shown in Sec. 3.
   (c) The pipeline for human-in-the-loop corrections of benchmark scenarios is shown in Sec. 4.
   (d) Example visualizations of task-oriented scene graphs and layouts are shown in Sec. 5.
4. **GENMANIP demonstration collection:**
   (a) Articulated objects assets and primitive skills teleoperation are described in Sec. 6.
   (b) Implementation details about BC data collection are included in Sec. 7.
5. **GENMANIP-BENCH:**
   (a) Implementation details about evaluating scene graph relations are shown in Sec. 8.
   (b) Implementation details and prompts about modular manipulation systems, including visual prompts and step-by-step implementation, are included in Sec. 9.
   (c) Implementation details of learning-based methods are shown in Sec. 10.
   (d) Failure case visualizations of the modular systems are shown in Sec. 11.
6. Characterizing the sim-to-real gap is discussed in Sec. 12.

1. Experimental Setups

GENMANIP operates within a tabletop scenario using a single Franka Arm on Isaac Sim 4.1.0. It features three distinct camera positions for prompt-based methods, along with two additional positions for data collection and testing of learning-based methods, as depicted in Figure A 1. By default, each camera captures RGB images and can optionally provide depth images and point clouds. We have designed user-friendly interfaces for easy adjustment of the cameras' intrinsic and extrinsic parameters. Users can also add or remove cameras to customize views and tailor benchmarks to their specific methods. In the GENMANIP-BENCH, we set the scene background to white to achieve a cleaner view and simplify the testing context.

Figure A - 1. Camera setups for modular manipulation systems and learning-based methods in GENMANIP-BENCH.

We encapsulate potential model calls (e.g., Anygrasp [1], SAM2 [4], custom learning-based models) as backend services, interacting with the main program. This architecture facilitates asynchronous communication between the models and the program, enhancing usability and reusability.

2. Prompts of ToSG Generation

The base prompt for generating the task-oriented scene graph is shown in Prompt 1. To generate four types of tasks, we incorporate task-specific prompts into the base prompt. Note that we do not include a long-horizon prompt, as this type of task can be derived from other tasks. The task specific prompts are listed in Prompt 2, 3 and 4.

Prompt 1: ToSG Generation Prompt

You are an assistant specializing in simulation scene design. Your task is to select at least [Num_of_objects] objects from the provided list (each with specific names and descriptions) to include in a simulation scene. Each object should be strategically positioned relative to at least one other object using the specified spatial orientations: left, right, front, back, and top.

Available Objects and corresponding states:
- Name: Porcelain Plate
UID: 008
Description: A round, white porcelain plate with intricate blue floral patterns.
States: None
// omitted
Your Task:
Design a task-oriented scene graph consisting of the following three parts:
1. **Instruction:** You need to design a tabletop manipulation task that will change the layout of the objects. If the task does not require changing the object's state, it should mimic a typical 'pick and place' activity as seen in daily life. However, if the task involves altering the object's state—such as opening or closing a cabinet—select an appropriate verb to accurately describe the action needed to modify the object's state.
- Focus on clear and functional interaction between the objects.
- When you need to refer to an object, you can use the provided name. You can change the name based on the instruction, but please do not create any ambiguity.
- The goal condition is sufficient to judge the instruction.
- [Task specified prompt]
2. **Goal Conditions:** Define the objectives of the Instruction, specifying:
- The names and unique identifiers (UIDs) of the objects involved.
- Their final states (choose from the given states; if none, state as 'none').
- The relative positions intended between the objects (using 'front', 'back', 'left', 'right', 'near', 'top').
- If multiple conditions are applied, they should be expressed in disjunctive normal form. Each atomic variable should determine whether a single spatial relation or object state is satisfied.
3. **Scene Graph:** Design a scene graph that captures the spatial relationships and states of the objects in the scene.
- Begin by describing the initial scene, emphasizing the spatial relationships among objects within a natural setting. This will help you write the scene graph.
- Each edge in the scene graph must have two objects.
- Include sufficient edges to represent all connections and interactions.
Note: Assume all objects are on the table by default, so avoid specifying 'top' or spatial relations with it. The instruction must alter the initial layout, and the goal condition should not match the initial scene graph. Avoid the common mistake of creating a circular transformation where reversing the objects' positions results in the same spatial relationships (e.g., "A on the left of B" becoming "B on the right of A").
Output Format:
Return the output in the following JSON format:

```
{
    "instruction": "Instruction for the task", // within 30 words
    "goal_conditions": // disjunctive normal form: multiple conditions combined with AND inside [], connected by OR between [].
    Example structure provided below.
    [
        [
         {
            "obj1": "Name of the first object for the task",
            "obj1_uid": "UID of the first object",
            "obj1_state": "state of obj1",
            "obj2": "Name of the second object for the task, or 'none' if not needed",
            "obj2_uid": "UID of the second object, or 'none' if not needed",
            "position": "front/back/left/right/near/top, or 'none' if not needed"
         }
        ]
    ],
    "scene_graph": {
        "description": "Describe the initial scene layout, emphasizing the spatial relationships between objects.
        Include details about the objects involved and their specific positions relative to one another.",
        "edges": [
            {
                "obj1": "Name of the first object",
                "obj1_uid": "UID of the first object",
                "position": "front/back/left/right/near/top",
                "obj2": "Name of the second object", // can not be None
                "obj2_uid": "UID of the second object"
            }
            // ...
        ],
        "nodes": [
            {
                "obj_uid": "UID",
                "state": "initial state"
            }
            // ...
        ]
    }
}
```

Prompt 2: Spatial Reasoning Task Prompt

Design an instruction that assesses a model's spatial reasoning ability, which is the capability to understand, interpret, and infer indirect spatial relationships among objects to determine their precise locations. The instruction must clearly describe a task that involves interpreting complex spatial configurations.

Prompt 3: Appearance Reasoning Task Prompt

Design an instruction that evaluates a model's ability to reason about and recognize visual attributes such as color, size, shape, material, and other distinctive characteristics. The instruction must use these visual traits to uniquely identify objects without relying on their names or identifiers.

Prompt 4: Common Sense Reasoning Task Prompt

Design an instruction to evaluate a model's common sense reasoning. Present a clear, everyday scenario that involves a practical problem or goal related to human needs. The model should apply basic principles like cause and effect or logical outcomes to choose one specific action that solves the problem.

3. Pseudo code of Layout Generation based on Scene Graph

The pseudo code for the layout generation pipeline is presented in Algorithm 1.

Algorithm 1 Layout construction pipeline.

Input: Scene Graph G from task-oriented scene graph
Output: Generated Layout L
Initialize $L \leftarrow \emptyset$
$O \leftarrow \text{TopologicalSort}(G)$
$C \leftarrow \text{ExtractRelationalConstraints}(G)$
for o_i in O **do**
  $T_i \leftarrow \text{GetTopologicalLevel}(o_i, G)$
  while True **do**
    $P_i \leftarrow \text{FindPlacement}(o_i, T_i, L, C)$
    if $\text{ValidatePlacement}(P_i, o_i, C, L)$ **then**
      $L \leftarrow L \cup (o_i, P_i)$
      Break
    end if
  end while
end for
return L
Subroutine Definitions:
$\text{TopologicalSort}(G)$: Sort objects in G based on 'on' and 'in' relations.
$\text{ExtractRelationalConstraints}(G)$: Extract relational constraints such as "nearby" from G.
$\text{GetTopologicalLevel}(o, G)$: Get the topological level of object o from G.
$\text{FindPlacement}(o, T, L, C)$: Calculate placement of o based on level T, layout L, and constraints C.
$\text{ValidatePlacement}(P, o, C, L)$: Validate placement P of object o against C and L.

4. Human-in-the-Loop Corrections of GenManip-Bench Scenarios

We begin by sampling four instruction types from the generated corpus to ensure comprehensive coverage of object appearances, common-sense knowledge, spatial relationships, and long-horizon tasks. Human annotators, with privileged access to the scene graph and USD files, use IsaacSim for detailed inspections. They also utilize a tagged, organized object collection

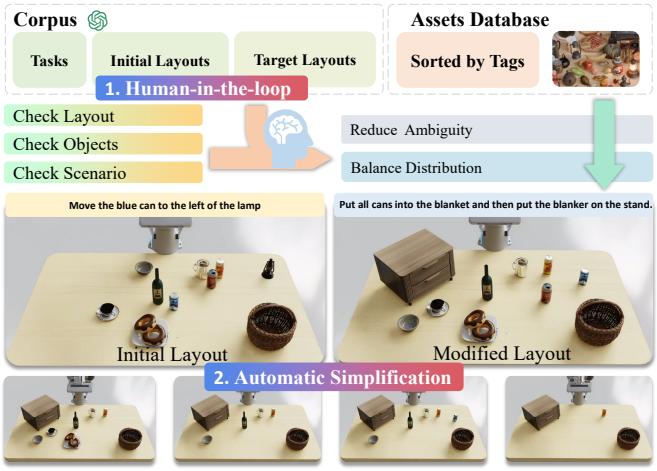

Figure A - 2. **Human-in-the-Loop corrections of benchmark scenarios.**

for asset retrieval. Subsequently, annotators refine layouts, objects, and scenarios based on GPT-generated data to ensure daily consistency. They evaluate instruction feasibility, identifying potential issues such as collisions—for example, a robot grasping a blue can that interferes with a nearby bottle. Finally, to establish a balanced benchmark across generalization dimensions, we meticulously review the distribution of instructions and select 200 task scenarios for benchmarking.

5. Example Visualization of ToSG and Layout

We visualize example task-oriented scene graphs and their respective tabletop layouts in Figure A- 3.

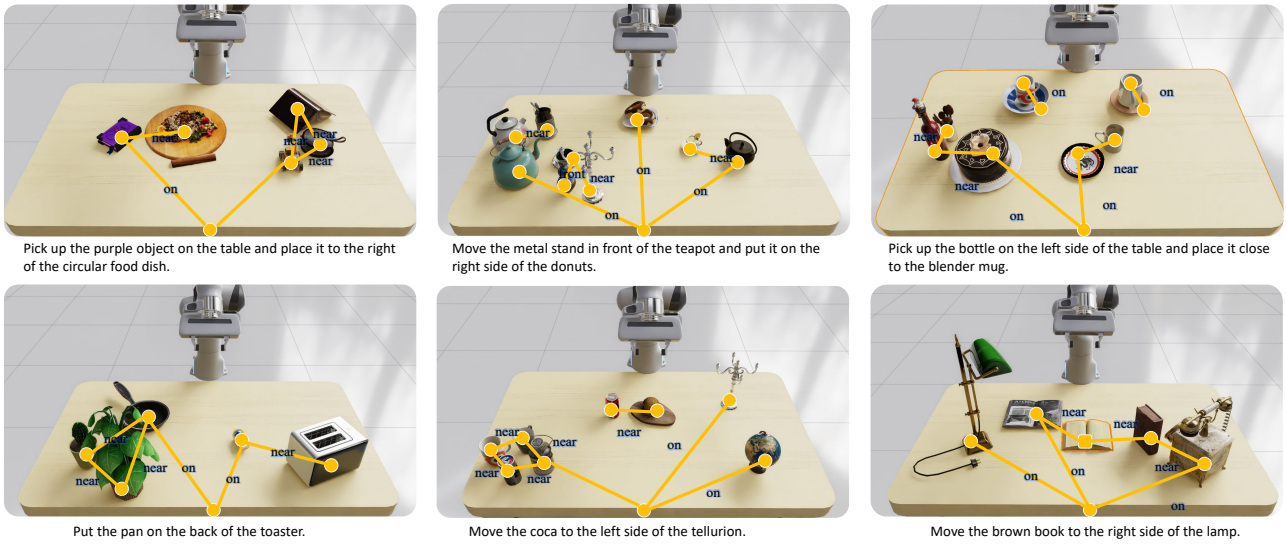

Figure A - 3. Example visualization of task-oriented scene graph and layouts.

6. Articulated Objects And Teleoperation for Primitive Skills

Figure A-4 displays several of these objects in Isaac Sim, which require specific manipulations. Following the approach of Mimicgen [3], we collect primitive skills from human teleoperation trajectories for these articulated objects, as illustrated in Figure A- 5.

Figure A - 4. **Articulated objects.** The 6/100 articulated objects with different manipulation strategies are shown.

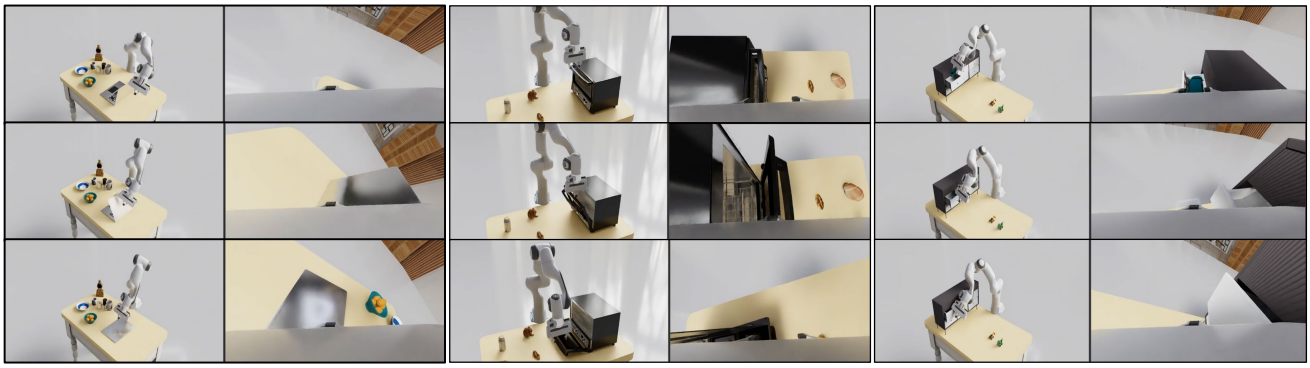

Figure A - 5. **Teleoperation.** Human annotators utilize the SpaceMouse, a 6-DoF device, to teleoperate the Franka robot in IsaacSim.

7. Implementation Details about Behavior-Cloning Data Collection

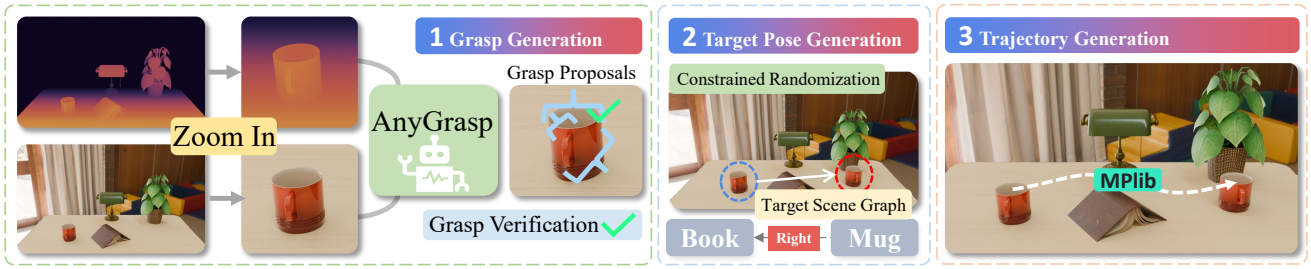

Figure A - 6. **BC data collection pipeline.** The pipeline comprises three steps: (1) grasp generation using AnyGrasp [1], (2) target position generation constrained by the scene graph, and (3) trajectory generation with MPlib [2].

The BC data collection pipeline is shown in Figure A- 6. First, AnyGrasp [1] is utilized to generate the grasp pos of the end effector. To ensure high-quality grasp proposals, we crop the target object from the rendered RGB-D image and filter the grasp by direction to avoid potential collision between the end effector and the table. Then we randomly place the target object based on the goal scene graph. Finally, a trajectory is generated to connect smoothly the Franks initial pose, grasp pose, and target place pose with the help of MPlib [2]. Each demonstration is tested in the physics-based simulator to ensure they can be implemented in the real world.

Specifically, for data collection, Isaac Sim is configured to render and perform physics calculations at 60 FPS, using the `plan_pose` function from MPlib for motion planning, with `rrt_range` set to 0.001. We use the PD Controller provided

by Isaac Sim for force control of the Franka robot. Data collection involves recording RGB-D images from two camera perspectives. The initial resolution is 1280×720, with depth in meters, and these images are resized to 512×288. The color images are saved as JPG files, while the depth images, after being multiplied by 1000, are saved as PNG files. We choose the absolute joint positions as the representation of Franka's pose and record Franka's state and action at each frame. Additionally, we process the absolute joint positions into end effector coordinates in the robot's coordinate system and Euler angles in XYZ order as another form of representation.

8. Evaluation Details

To evaluate object relationships, the final scenes post-method execution are converted into point clouds and the point clouds for each object are extracted. In this section, we detail the method for determining spatial relationships among these point clouds, encompassing horizontal, vertical, and multi-object interactions. The pseudo-code for assessing these spatial relationships is presented in Algorithm 2.

Horizontal relations. Horizontal relations between two point clouds are evaluated based on their relative positions in the XY plane. The steps are as follows:

1. Compute the 2D distance between the bounding boxes of the point clouds.
2. If the distance is greater than a threshold `XY_DISTANCE_CLOSE_THRESHOLD`, the point clouds are considered separate, and further horizontal relation analysis is skipped.
3. If the distance is small enough, check for overlap along both the X and Y axes. The possible horizontal relationships are:
   - **Left-Right**: This relationship is determined when the point clouds overlap along the X-axis but not the Y-axis. The objects are positioned side by side along the X-axis.
   - **Front-Back**: This relationship is determined when the point clouds overlap along the Y-axis but not the X-axis. The objects are positioned side by side along the Y-axis.
   - **Near**: If there is overlap in both the X and Y axes, the point clouds are considered to be near each other.

Vertical relations. Vertical relations between two point clouds are evaluated based on the Z-axis distance and their relative positions in 3D space. The key steps are:

1. Calculate the vertical distance between the point clouds, including the distance from the top of one object to the bottom of the other.
2. If the point clouds are close enough (within a threshold distance `MAX_TO_BE_TOUCHING_DISTANCE`), the function evaluates if one object is:
   - **On/Beneath**: One object is on top of or below the other.
   - **Supporting/Supported by**: If the objects are in contact or near each other, the function checks if one object supports the other based on the overlap area ratio.

Multi-Object Relations. When a third point cloud `point_cloud_c` is provided, the function computes the centroids of the point clouds and evaluates the angular relationship between the vectors formed by the centroids. The steps are as follows:

1. Compute the centroid of each point cloud: `anchor1_center`, `anchor2_center`, and `target_center`.
2. Construct vectors `vector1` and `vector2` from the target centroid to each of the anchor centroids.
3. Normalize the vectors and compute the cosine of the angle between them.
4. If the angle is smaller than a predefined threshold `ANGLE_THRESHOLD`, the relationship is labeled as `between`, indicating that the target object is positioned between the two anchor objects.

Algorithm 2 Infer Spatial Relationship Between Point Clouds

Input: Point clouds P_{A}, P_{B}, (optional P_{C}), bounding boxes $\min A$, $\max A$, $\min B$, $\max B$
Output: Spatial relationship between A and B (or C)
Constants: Constants for between, in/out of, above/below, on/beneath, near, left/right, front/back.

if P_{C} exists **then**
  $\vec{AB} \leftarrow \mathrm{CalculatePointCloudVector}(\mathrm{GetCenter}(P_A), \mathrm{GetCenter}(P_B))$
  $\vec{BC} \leftarrow \mathrm{CalculatePointCloudVector}(\mathrm{GetCenter}(P_B), \mathrm{GetCenter}(P_C))$
  Angle $\leftarrow \mathrm{CalculateAngleByVectors}(\vec{AB}, \vec{BC})$
  if IS_BETWEEN(Angle) **then**
    return "between", "between"
  end if
else
  $d_{\mathrm{xy}} \leftarrow \mathrm{CalculateXYDistance}(P_{\mathrm{A}}, P_{\mathrm{B}})$
  if IS_INTOUCH(d_{xy}) **then**
    if IS_INSIDE(P_{A}, P_{B}) **then**
      return "in", "out of"
    else if IS_INSIDE(P_{B}, P_{A}) **then**
      return "out of", "in"
    end if
    if IS_OVERLAP(P_{A}, P_{B}, axis=z) **then**
      if GetCenter(P_A)[z] $>$ GetCenter(P_B)[z] **then**
        return "on", "beneath"
      else if GetCenter(P_A)[z] $>$ GetCenter(P_B)[z] **then**
        return "beneath", "on"
      end if
    else if IS_OVERLAP(P_{A}, P_{B}, axis=x) and **not** IS_OVERLAP(P_{A}, P_{B}, axis=y) **then**
      if GetCenter(P_A)[x] $>$ GetCenter(P_B)[x] **then**
        return "front", "back"
      else if GetCenter(P_A)[x] $<$ GetCenter(P_B)[x] **then**
        return "back", "front"
      end if
    else if IS_OVERLAP(P_{A}, P_{B}, axis=y) and **not** IS_OVERLAP(P_{A}, P_{B}, axis=y) **then**
      if GetCenter(P_A)[y] $>$ GetCenter(P_B)[y] **then**
        return "left", "right"
      else if GetCenter(P_A)[y] $<$ GetCenter(P_B)[y] **then**
        return "right", "left"
      end if
    end if
    return "near", "near"
  end if
end if
return No Relationship
Subroutine Definitions:
GetCenter: Calculates the center point of the point cloud.
CalculatePointCloudVector: Obtains the vector through the center point of a point cloud.
CalculateAngleByVectors: Computes the angle between two vectors.
CalculateXYDistance: Determines the shortest distance between the projections of two point clouds on the XY plane.
IS_BETWEEN/IS_INTOUCH/IS_INSIDE: Check if the input values satisfy the constraints defined by the respective relationships.
IS_OVERLAP: Verifies whether the projection overlap of two point clouds along a specified coordinate axis meets the defined constraints.

9. Implementation Details and Prompts about Modular Manipulation System

In this section, we present the complete prompts of modular manipulation system. Specifically, we first use the prompts from the task decomposition by Prompt 5 to divide the task, then use SoM by Prompt 6 for mask selection. The CtoF SoM by Prompt 7 is an optional component used to switch between different modules. After that, we run grab point selection by Prompt 8 to obtain the grab point and execute grid-based path planning by Prompt 9 or point-to-point path planning by Prompt 10 to acquire a series of waypoints. At this stage, we have obtained a series of two-dimensional coordinates in the image coordinate system, including the grab point and waypoints.

In the simulation evaluation with Isaac Sim, the prompt baseline acquires transformation-related information through the API for obtaining the camera's intrinsic and extrinsic parameters. It then uses reprojection to obtain a series of three-dimensional coordinates in the world coordinate system. Additionally, it calls Anygrasp [1] and uses the distance threshold between the grasp pose and the 3D grab point, along with score-based sorting, to determine the grasp pose.

Subsequently, the prompt baseline uses MPlib [2] for motion planning, generating the joint positions from the grasp pose to each waypoint. It then uses a PD controller in Isaac Sim to perform force control, thereby controlling the Franka robot to execute the operation. After each execution of the modular methods, the Task completion checker by Prompt 11 is called to check the task completion status and achieve a closed-loop system.

Due to existing limitations in simulation skills, trajectory-action chaining currently only supports pick-and-place actions. However, the modular system retains its extensibility and can be equipped with more complex functionalities in the future. Additionally, the modular system is not coupled with Isaac Sim. In fact, the modular baseline only requires the camera's intrinsic and extrinsic parameters to obtain three-dimensional trajectory points. By deploying it as a service, it supports high concurrency, making batch evaluation possible.

Prompt 5: Task Decomposition

You are an assistant that can split complex pick and place tasks into simpler subtasks. Analyze the following task instruction and divide it into a list of subtasks.
Task instruction: {*instruction*}.
Return the subtasks in JSON format as a list. If the task is simple and does not require splitting, return a list with a single subtask.
Each subtask should consist of a single pick and place operation.
It is normal for most tasks to have only one subtask.
Example output:

```
{
    "subtasks":[
        "Pick the red apple and place it on the table.",
        "Pick the blue cup and place it next to the apple."
    ]
}
```

[Original Image]

Prompt 6: SoM

You are an assistant designed for creating pick and place operations. The image shows a scene currently observed by the camera, and another image segmented the original image into different parts using a segmentation model with annotations on the corresponding masks. You need to understand the task instructions and select the object required for the current task.
Task instruction: {*instruction*}.
Based on the above information, output the selected object in JSON format. If none of the masks include the object you need to select, output the number as -1 and the object name as "not_found.
Example output:

```
{
  "number": 3,
  "object_name": "apple",
  "color": "red"
}
```

[Original Image][Mask Annotated Image]

Prompt 7: Coarse-to-fine SoM

You are an assistant designed for fine-grained part picking operations. The image shows the object that needs to be operated on, and another image segmented the original image into different parts using a segmentation model with annotations on the corresponding masks. You need to understand the task instructions and select the specific part of the object that needs to be picked.
Task instruction: {*instruction*}.
Based on the above information, output the selected part in JSON format. If none of the masks include the part you need to select, output the number as -1 and the part name as "not_found."
Example output:

```
{
    "number": 3,
    "part_name": "handle",
    "color": "red"
}
```

[Croped Original Image][Croped Mask Annotated Image]

Prompt 8: Grab Point Selection

You need to grab the object: {*object_name*}. The image below are the original image and five sampled points. Please select the most appropriate grab point by specifying the point number (1-5) and provide a brief reason. Output the result in JSON format as follows:

```
{
  "selected_point": 3,
  "reason": "Point 3 provides the most stable grip based on the object's orientation."
}
```

[Original Image][Point Annotated Image]

Prompt 9: Grid-based Path Planning

You are a motion planning agent. For the task I provide, you need to plan a series of waypoints and guide the robotic arm to the final position to complete the task.
For the task {*instruction*}, with the previously identified object {*object_name*} and the grab points (marked in the image), you need to output the following series of details as described.
The whole path should begin with start point {*start_point*}
Each movement should include the grid cell number, the height above the table (0.1 to 0.5 meters), and the claw's orientation.
Output the list in JSON format where each item contains:
- grid_number: e.g., 'A1'
- height_m: float value between 0.1 and 0.5
- claw_orientation: one of ['up', 'down', 'left', 'right', 'front', 'back']
Example Output:

```
{
    "path":[\n'
        {"grid_number": "A1", "height_m": 0.3, "claw_orientation": "down"},
        {"grid_number": "B3", "height_m": 0.2, "claw_orientation": "front"},
        ...
    ]
}
```

[Original Image][Grid Annotated Image]

Prompt 10: Point-to-Point Path Planning

You are an assistant that selects a single point for the robot to move to. For the task {*instruction*}, with the previously identified object {*object_name*} and the grab points (marked in the image), you need to output the grid cell label to place the object at.
Based on the original image and the grid overlay, please select a grid cell label that represents the target point (where to place the object) for the robot.
Output the selected point in JSON format as follows:

```
{
    "selected_point": {"type": "grid", "label": "C4"}
}
```

[Original Image][Grid Annotated Image]

Prompt 11: Task completion checker

You are an assistant that checks whether a task is completed. For the given instruction, the provided image describes the current scene, and you need to descibe the given scene and determine whether the task is finished.
Task instruction: {*instruction*}.
Example output:

```
{
    "scene_description": "The description of the scene",
    "finished": true or false
}
```

[Original Image]

10. Implementation Details about Learning-based Models

Both GR-1 and ACT are trained on 1K trajectories per setting. For cross-scene training, each scene provides 1K episodes. The model predicts images from static and gripper cameras, forecasts the next 3 steps, and executes 1 step, using an input sequence length of 10. Training uses dropout, AdamW with cosine decay, a batch size of 512, and a learning rate of 2e-3. GR-1 is trained from scratch without pre-training.

11. Failure Case Visualization of Modular Manipulation System

We present a detailed analysis of agent failures, as illustrated in Figure A- 7. Tasks are categorized into four types: spatial, appearance, common sense, and long horizon. Grasp Failures are classified into three subcategories: (1) Grounding Error, resulting from incorrect masks generated by the Scene Object Model (SoM); (2) Grasp Proposal Error, caused by ungraspable poses identified by Context-to-Frame (CtoF) SoM or AnyGrasp; and (3) Grasp Error, arising from collisions or other operational issues. Similarly, Planning Failures are divided into: (1) Grounding Error, due to selecting an incorrect anchor object; (2) Location Obfuscation, involving confusion over placement specifications (*e.g.*, distinguishing "on" from "near"); and (3) Motion Planning Error, where the planned route cannot be executed.

The six representative failure cases are presented in Figure A- 8. These examples highlight the limitations of the current prompt-based baseline, revealing areas for improvement such as selecting more optimal grasp poses

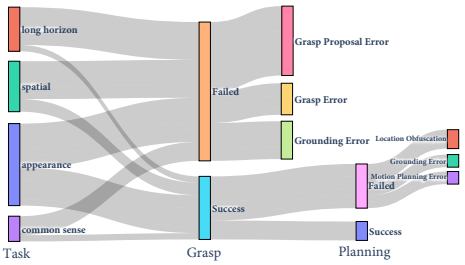

Figure A - 7. **Analysis of Failure Cases.** The figure categorizes failures by task type and identifies causes within grasp and planning.

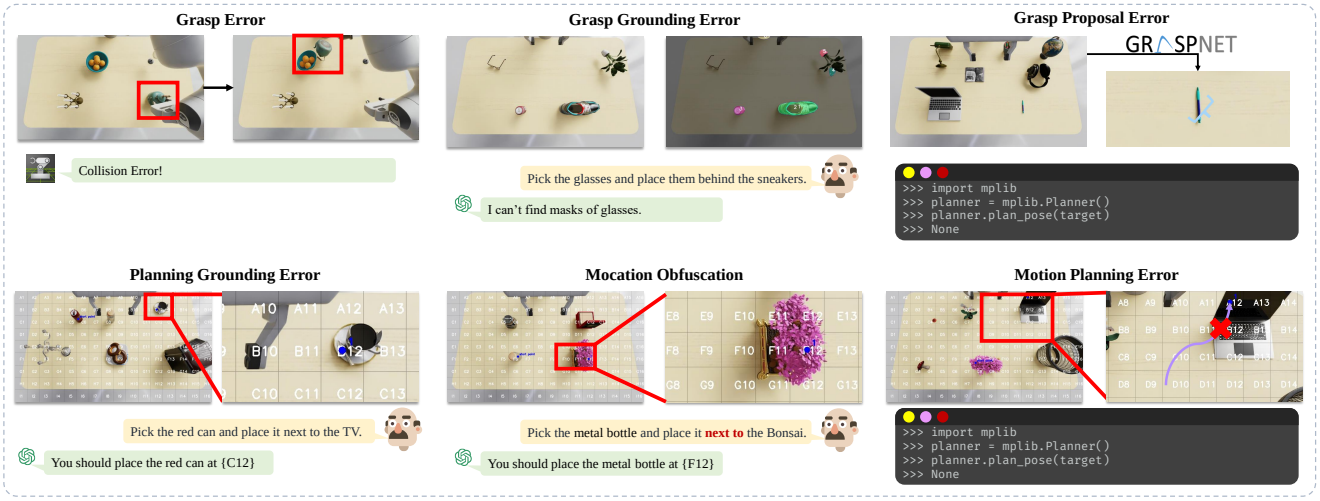

Figure A - 8. **Failure Cases Visualization.** Grasp Failures are classified into three subcategories: (1) Grounding Error, resulting from incorrect masks generated by the Scene Object Model (SoM); (2) Grasp Proposal Error, caused by ungraspable poses identified by Context-to-Frame (CtoF) SoM or AnyGrasp; and (3) Grasp Error, arising from collisions or other operational issues. Similarly, Planning Failures are divided into: (1) Grounding Error, due to selecting an incorrect anchor object; (2) Location Obfuscation, involving confusion over placement specifications (e.g., distinguishing "on" from "near"); and (3) Motion Planning Error, where the planned route cannot be executed.

and accurately grounding the destination. Addressing these challenges will require advancements in future methods, such as incorporating a stronger grasp proposal model, adopting a more robust chain of thought, or implementing closed-loop reasoning to enhance the accuracy and stability of the model's outputs.

12. Characterizing Sim-to-Real Gap

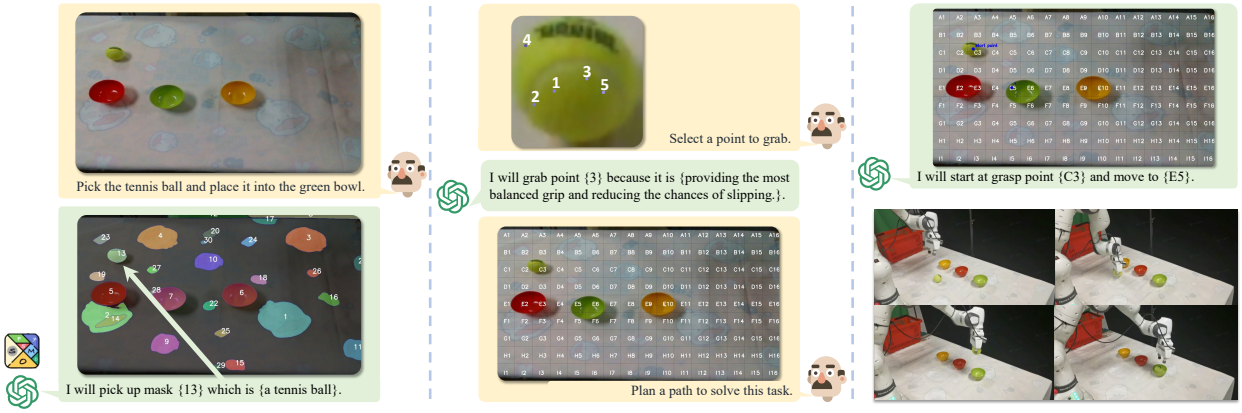

Figure A - 9. **Real-world deployment of modular manipulation system.**

To demonstrate the realism of our environment, we deployed the prompt-based modular methods on a Franka robot in real-world scenarios. The chosen result, shown in Figure A- 9, indicates that the baseline operates smoothly in the real environment with a measurable success rate. This suggests that our configured Isaac Sim environment closely mirrors the real world. Due to the costs associated with constructing real-world scenes, we leave further comparison between simulated and real environments as future work.

Despite the advancements achieved with GENMANIP in Isaac Sim, there still exists a notable sim-to-real gap between our simulation and the real-world environment. This gap manifests in the following two aspects:

- **Physical simulation**: Although Isaac Sim allows for the setting of physical parameters, including dynamic friction, static friction, restitution, density, etc., these settings demand extensive debugging and more detailed object annotations. Inappropriate friction coefficients may result in objects sticking or slipping during grasping, and unrealistic mass density can cause deviations in collision outcomes.
- **Lighting simulation**: Isaac Sim provides an interface for setting material parameters, including roughness, metallic, ior, thin-walled, etc., and the objects themselves contain various materials. However, inconsistencies in object sources can lead to unrealistic settings, such as varying texture resolutions and reflectivity, potentially resulting in unrealistic simulation scenes.

References

[1] Hao-Shu Fang, Chenxi Wang, Hongjie Fang, Minghao Gou, Jirong Liu, Hengxu Yan, Wenhai Liu, Yichen Xie, and Cewu Lu. Anygrasp: Robust and efficient grasp perception in spatial and temporal domains. *IEEE Transactions on Robotics*, 2023. 1, 5, 8

[2] Runlin (Kolin) Guo, Xinsong Lin, Minghua Liu, Jiayuan Gu, and Hao Su. MPlib: a Lightweight Motion Planning Library, 2023. 5, 8

[3] Ajay Mandlekar, Soroush Nasiriany, Bowen Wen, Iretiayo Akinola, Yashraj Narang, Linxi Fan, Yuke Zhu, and Dieter Fox. Mimicgen: A data generation system for scalable robot learning using human demonstrations. *arXiv preprint arXiv:2310.17596*, 2023. 4

[4] Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, Eric Mintun, Junting Pan, Kalyan Vasudev Alwala, Nicolas Carion, Chao-Yuan Wu, Ross Girshick, Piotr Dollár, and Christoph Feichtenhofer. Sam 2: Segment anything in images and videos. *arXiv preprint arXiv:2408.00714*, 2024. 1