

VLM-Grounder: A VLM Agent for Zero-Shot 3D Visual Grounding

Supplementary Material

1	A Dynamic Stitching	1
2	B Visual-Retrieval Benchmark Settings	1
3	C Ablation on Detectors	3
4	D VLM-Grounder Prompts	3
5	E Qualitative Results	5
6	F Discussions of Limitations	5

7 **A Dynamic Stitching**

8 We employ a dynamic stitching algorithm to organize images into various layouts, with the pseu-
9 docode provided in Algorithm 1. The process begins by calculating the largest layout that should be
10 used. Given an image sequence with n images, and a maximum number of stitched images L , we
11 first compute the quantity of each layout. We use the variables n_4 , n_8 , n_{16} , and n_{27} to represent the
12 number of (4, 1), (2, 4), (8, 2), and (9, 3) layouts, respectively.

13 For example, assuming $n = 84$ and $L = 6$, we know $n \leq 16L$. First, we calculate the minimum
14 number of (8, 2) layouts required. Each (8, 2) layout accommodates 8 more images than a (2, 4)
15 layout, so we divide the number of images exceeding what six (2, 4) layouts can store by 8 to find
16 the minimum number of (8, 2) layouts needed. In this example, it is 5. Next, we compute the layout
17 needed for the remaining images. We update the remaining image count to $84 - 5 * 8 * 2 = 4$ and
18 the stitched image count to $6 - 5 = 1$. Similarly, we determine that we need zero (2, 4) layouts and
19 one (4, 1) layout for the remaining images. Thus, we have determined the number of each layout
20 required. We then generate the stitched images in ascending order of layout size to ensure that only
21 the largest layout may have unused space, thereby minimizing resolution waste.

22 It is important to note that if the number of images is too large to be accommodated by L images
23 of the largest layout, we select the largest layout to minimize the total number of stitched images.
24 For any excess images, we maximize utilization efficiency by invoking the `dynamic_stitching`
25 function again to find the appropriate layout, setting the fixed number to 1 to minimize the count of
26 stitched images. In this case, we first generate (9, 3) layouts and then recursively call the function to
27 generate the remaining layouts, which may result in some unused space in smaller layouts.

28 **B Visual-Retrieval Benchmark Settings**

29 We randomly selected 1,000 images from the ScanNet dataset, assigning each a unique ID ranging
30 from 00000 to 00999. Each image ID was annotated in red at the top-left corner. Additionally, a
31 color block was generated at a random position within each image, using one of six colors: red,
32 green, blue, yellow, white, or black. The images were then stitched using specific layouts, forming
33 the basic image sets sent to the VLM. The VLM’s task was to identify all images, retrieve their
34 IDs, and determine the color of the blocks. The VLM was required to return two lists—IDs and
35 corresponding colors—as demonstrated in Fig.3. of the main paper.

Algorithm 1: Dynamic Stitching Algorithm

```
1 Function dynamic_stitching(imgs, L):  
   // candidate_layouts: (4, 1), (2, 4), (8, 2), (9, 3)  
   Input: image sequence imgs, the maximum number of stitched images L  
   Output: stitched image sequence res  
2   n ← len(imgs);  
3   res ← [];  
4   if n ≤ 4L then                                     // (4, 1) layout is enough  
5     | res ← stitch_image(imgs, (4, 1));  
6   else if n ≤ 8L then                                   // at least one (2, 4) layout is used  
7     | n8 ← ⌈(n − 4L)/4⌉;  
8     | n4 ← L − n8;  
9     | res ← res + stitch_image(imgs[0 ... 4n4 − 1], (4, 1));  
10    | res ← res + stitch_image(imgs[4n4 ... ], (2, 4));  
11  else if n ≤ 16L then                                   // at least one (8, 2) layout is used  
12    | n16 ← ⌈(n − 8L)/8⌉;  
13    | n ← max(n − 16n16, 0);                               // number of images remaining  
14    | n4,8 ← L − n16;                                       // number of (4, 1), (2, 4) layouts  
15    | n8 ← ⌈(n − 4n4,8)/4⌉;  
16    | n4 ← n4,8 − n8;  
17    | res ← res + stitch_image(imgs[0 ... 4n4 − 1], (4, 1));  
18    | res ← res + stitch_image(imgs[4n4 ... 4n4 + 8n8 − 1], (2, 4));  
19    | res ← res + stitch_image(imgs[4n4 + 8n8 ... ], (8, 2));  
20  else if n ≤ 27L then                                   // at least one (9, 3) layout is used  
21    | n27 ← ⌈(n − 16L)/11⌉;  
22    | n4,8,16 ← L − n27;                                     // number of (4, 1), (2, 4), (8, 2) layouts  
23    | n ← max(n − 27n27, 0);                               // number of images remaining  
24    | n16 ← ⌈(n − 8n4,8,16)/8⌉;  
25    | n4,8 ← n4,8,16 − n16;                                       // number of (4, 1), (2, 4) layouts  
26    | n ← max(n − 16n16, 0);  
27    | n8 ← ⌈(n − 4n4,8)/4⌉;  
28    | n4 ← n4,8 − n8;  
29    | res ← res + stitch_image(imgs[0 ... 4n4 − 1], (4, 1));  
30    | res ← res + stitch_image(imgs[4n4 ... 4n4 + 8n8 − 1], (2, 4));  
31    | res ← res + stitch_image(imgs[4n4 + 8n8 ... 4n4 + 8n8 + 16n16 − 1], (8, 2));  
32    | res ← res + stitch_image(imgs[4n4 + 8n8 + 16n16 ... ], (9, 3));  
33  else                                                     // use more than L stitched images  
34    | n27 ← ⌊n/27⌋;  
35    | res ← res + stitch_image(imgs[0 ... 27n27 − 1], (9, 3));  
36    | res ← res + dynamic_stitching(imgs[27n27 ... ], 1);  
37  return res;
```

Table 1: **3D Visual Grounding Results with YOLOv8-World and Grounding DINO 1.5.** * indicates that the evaluation is based on 2D masks.

Methods	Overall		Unique		Multiple	
	Acc@0.25	Acc@0.5	Acc@0.25	Acc@0.5	Acc@0.25	Acc@0.5
VLM-Grounder (YOLOv8-World)	44.8	28.4	57.5	31.9	41.9	27.6
VLM-Grounder (GDINO-1.5)	51.6	32.8	66.0	29.8	48.3	33.5
VLM-Grounder* (YOLOv8-World)	53.2	45.2	74.5	63.8	48.3	40.9
VLM-Grounder* (GDINO-1.5)	62.4	53.2	87.2	76.6	56.7	47.8

Occasionally, the VLM might retrieve the same ID from different images, leading to conflicts where multiple ID-color pairs exist for the same ID. In such cases, if at least one retrieved ID matches the ground truth, it is considered correct. In other words, we calculated the Recall as the accuracy in this benchmark. For instance, in Fig.3. of the main paper, if four images were input and the VLM retrieved four pairs, but the pair 00003-yellow was incorrect (the ground truth being 00003-blue), the accuracy for this benchmark would be 0.75.

The benchmark investigated two primary variables:

Stitching layout. The stitching layout defines the rows and columns in which images are stitched, which can be regarded as “visual resolution”.

Visual length. The number of stitched images included in a single conversation, which can be regarded as “visual context length”.

We also measured the request time cost. By duplicating an image from 1 to 30 times within a request, we conducted 10 trials for each duplication count and calculated the average request time cost.

C Ablation on Detectors

As Grounding DINO-1.5 [1] is a closed-source model, we can only request detections through its API. For open-source research, we also employ the widely-used open-source alternative YOLOv8-World [2, 3] for our experiments. Results on the ScanRefer [4] dataset are presented in Tab. 1.

D VLM-Grounder Prompts

We used several prompts in our work, as shown in the Tab. 2, including query_analysis_prompt, grounding_system_prompt, input_prompt, bbox_select_prompt, image_ID_invalid_prompt, and detection_not_exist_prompt.

For each query, we utilize the query_analysis_prompt to extract the category and associated conditions of the target object, such as position, shape, color, or relative relationships with other objects. In the grounding and feedback process, we first employed the grounding_system_prompt to guide VLM in performing visual grounding tasks. Then, we utilize the input_prompt to provide information such as our image, query statement, target object category, and grounding conditions, with stitched images appended. VLM would return the query results in the specified JSON format.

If the target image ID in the returned results does not contain any target object, we use the detection_not_exist_prompt to inform VLM and request it to make a new selection. In case the image ID provided cannot find the corresponding image, we employ the image_ID_invalid_prompt to notify VLM for a fresh selection. Furthermore, if there are multiple target objects in the chosen image, we use the bbox_select_prompt to instruct VLM in selecting the correct bounding box ID.

Table 2: **Prompts of VLM-Grounder.** The placeholders in the table represent different variables. `{query}` denotes the user query, while `{pred_target_class}` and `{conditions}` represent the target object’s category and grounding conditions, respectively. `{num_view_selections}` refers to the total number of images, and `{num_candidate_bboxes}` indicates the number of candidate bounding boxes. In the `image_ID_invalid_prompt` and `detection_not_exist_prompt`, `{image_id}` refers to the image ID selected by the VLM.

query_analysis_prompt

You are working on a 3D visual grounding task, which involves receiving a query that specifies a particular object by describing its attributes and grounding conditions to uniquely identify the object. Here, attributes refer to the inherent properties of the object, such as category, color, appearance, function, etc. Grounding conditions refer to considerations of other objects or other conditions in the scene, such as location, relative position to other objects, etc. Now, I need you to first parse this query, return the category of the object to be found, and list each of the object’s attributes and grounding conditions. Each attribute and condition should be returned individually. Sometimes the object’s category is not explicitly specified, and you need to deduce it through reasoning. If you cannot deduce after reasoning, you can use ‘unknown’ for the category. Your response should be formatted as a JSON object. Here are some examples:

Input:

Query: this is a brown cabinet. it is to the right of a picture.

Output:

```
{
  "target_class": "cabinet",
  "attributes": ["it's brown"],
  "conditions": ["it's to the right of a picture"]
}
```

...(two more examples)

Ensure your response adheres strictly to this JSON format, as it will be directly parsed and used.

Query: `{query}`

grounding_system_prompt

You are good at finding objects specified by user queries in indoor rooms by watching the videos scanning the rooms.

bbox_select_prompt

Great! Here is the detailed version of your selected image. There are `{num_candidate_bboxes}` candidate objects shown in the image. I have annotated each object at the center with an object ID in white color text and black background. Do not mix the annotated IDs with the actual appearance of the objects. Please give me the ID of the correct target object for the query. Reply using JSON format with two keys “reasoning” and “object_id” like this:

```
{
  "reasoning": "your reasons", // Explain the justification why you select the object ID.
  "object_id": 0 // The object ID you selected. Always give one object ID from the image, which you are the
  most confident of, even you think the image does not contain the correct object.
}
```

image_ID_invalid_prompt

The image `{image_id}` you selected does not exist. Did you perhaps see it incorrectly? Please reconsider and select another image. Remember to reply using JSON format with the three keys “reasoning”, “target_image_id”, and “reference_image_ids” as required before.

detection_not_exist_prompt

The image `{image_id}` you selected does not seem to include any objects that fall into the category of `{pred_target_class}`. Please reconsider and select another image. Remember to reply using JSON format with the three keys “reasoning”, “target_image_id”, and “reference_image_ids” as required before.

input_prompt

Imagine you are in a room and are asked to find one object. Given a series of images from a video scanning an indoor room and a query describing a specific object in the room, you need to analyze the images to locate the object mentioned in the query within the images. You will be provided with multiple images, and the top-left corner of each image will have an ID indicating the order in which it appears in the video. Adjacent images have adjacent IDs. Please note that to save space, multiple images have been combined into one image with dynamic layouts. You will also be provided with a query sentence describing the object that needs to be found, as well as a parsed version of this query describing the target class of the object to be found and the conditions that this object must satisfy. Please find the ID of the image containing this object based on these conditions. Note that I have filtered the video to remove some images that do not contain objects of the target class. To locate the target object, you need to consider multiple images from different perspectives and determine which image contains the object that meets the conditions. Note, that each condition might not be judged based on just one image alone. Also, the conditions may not be accurate, so it's reasonable for the correct object not to meet all the conditions. You need to find the most possible object based on the query. If you think multiple objects are correct, simply return the one you are most confident of. If you think no objects are meeting the conditions, make a guess to avoid returning nothing. Usually the correct object is visible in multiple images, and you should return the image in which the object is most clearly observed. Your response should be formatted as a JSON object with three keys "reasoning", "target_image_id", and "reference_image_ids" like this:

```
{
  "reasoning": "your reasoning process" // Explain the process of how you identified and located the target object.
  // If reasoning across different images is needed, explain which images were used and how you reasoned with them.
  "target_image_id": "00001", // Replace with the actual image ID (only one ID) annotated on the image that contains the target object.
  "reference_image_ids": ["00001", "00002", ...] // A list of IDs of images that are used to determine whether the conditions are met or not.
}
```

Here is a good example:

query: Find the black table that is surrounded by four chairs.

```
{
  "reasoning": "After carefully examining all the input images, I found image 00003, 00005, and 00021 contain different tables, but only the tables in image 00003 and 00021 are black. Further, I found image 00001, image 00002, image 00003, and image 00004 show four chairs and these chairs surround the black table in image 00003. The chair in image 00005 does not meet this condition. So the correct object is the table in image 00003",
  "target_image_id": "00003",
  "reference_image_ids": ["00001", "00002", "00003", "00004"]
}
```

Now start the task:

Query: "{query}"

Target Class: {pred_target_class}

Conditions: {conditions}

Here are the {num_view_selections} images for your reference.

68 E Qualitative Results

69 In this section, we present three demonstrations to elucidate the capabilities and behavior of VLM-Grounder
70 in various scenarios. First, in Fig. 1, we illustrate the basic execution process involving a single target object
71 within a scene. Subsequently, we demonstrate the execution process in a more complex scene containing
72 multiple target objects, where the VLM is employed to accurately select the correct object, as in Fig. 2. Lastly,
73 we showcase the execution process in a scenario where the VLM initially selects an incorrect image, thereby
74 triggering a feedback mechanism, as shown in Fig. 3. Morphological operations are applied to all the masks
75 including matched images. In all these examples, we only illustrate four ensemble images and show the result
76 of the morphological operation on the anchor mask. The system_prompt and query analysis are also omitted in
77 the figures for clarity.

78 F Discussions of Limitations

79 While VLM-Grounder achieves superior zero-shot 3D visual grounding by directly operating on 2D images
80 without requiring 3D point clouds or object priors, it has several limitations:

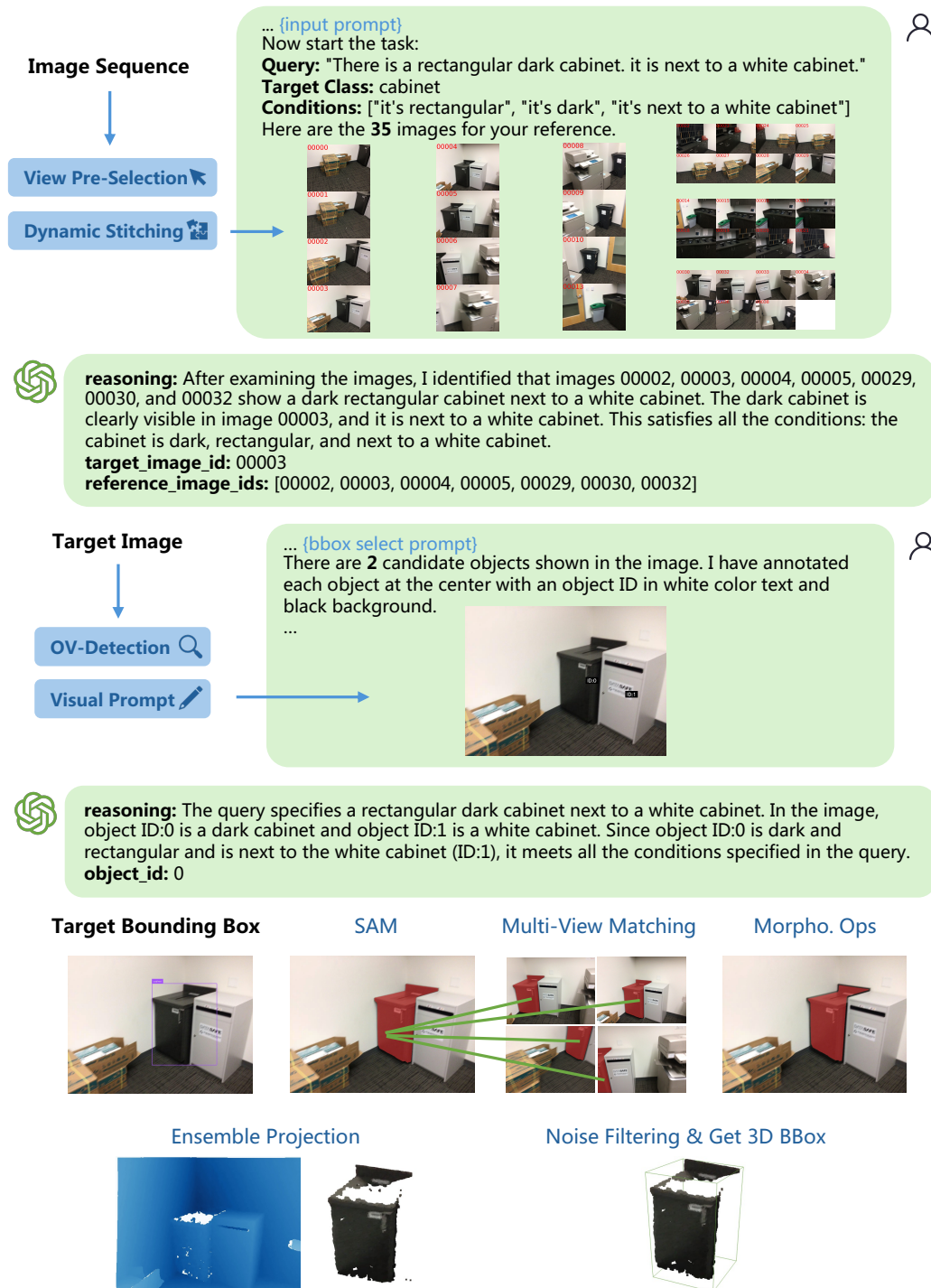
81 **Capabilities of VLMs.** VLM-Grounder depends on the vision-language model (VLM) for analyzing
82 grounding conditions and locating target objects in sequences of 2D images. If the VLM lacks the ability to
83 process multiple images or struggles with scene understanding from real 2D scans, performance may degrade.
84 In this study, we use the GPT-4o model, which delivers excellent results. VLM technology is continuously
85 advancing, and VLM-Grounder’s modular design allows us to replace the current VLM with more powerful
86 models as they become available, potentially enhancing future performance.

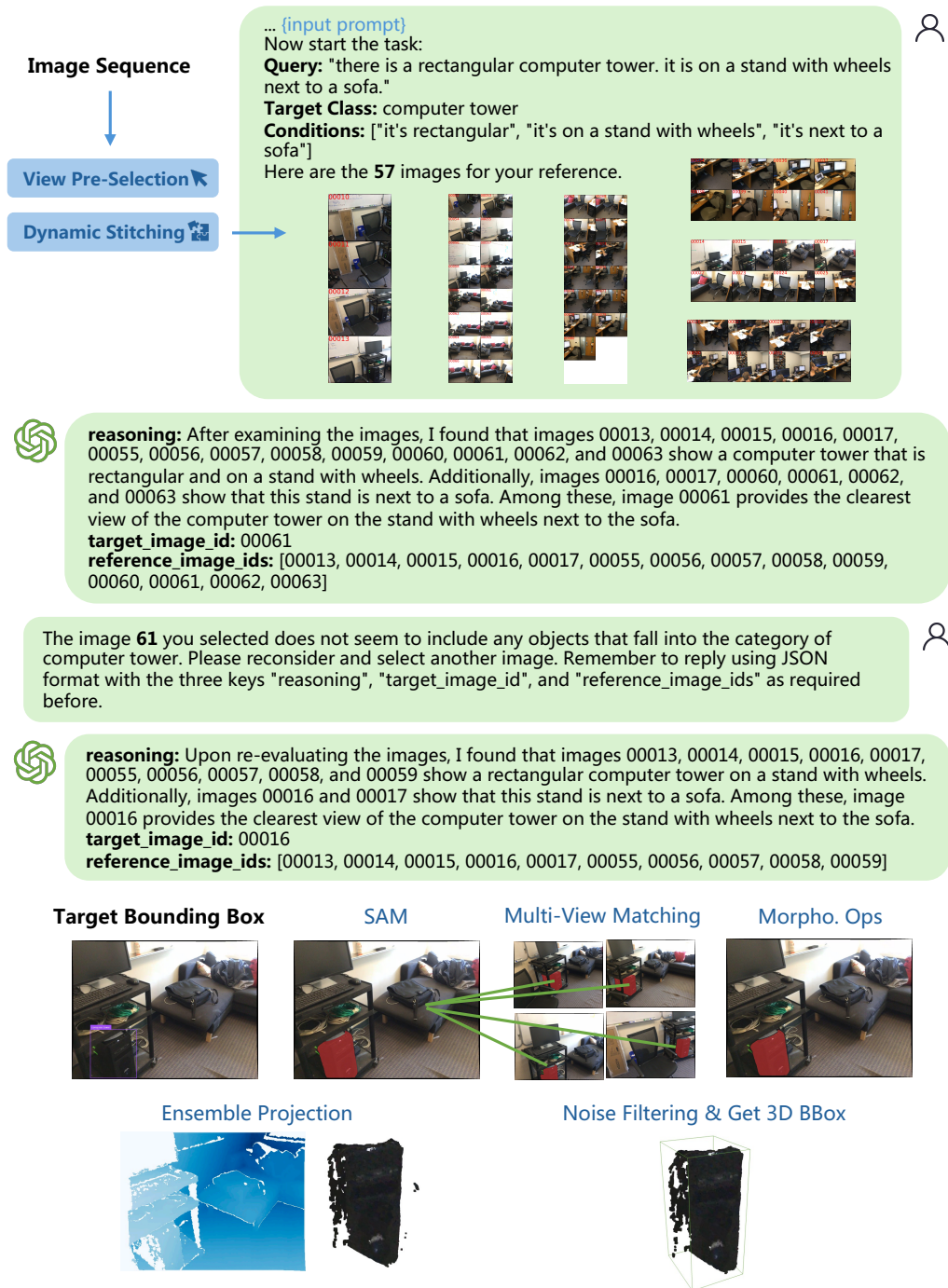
87 **Noise from 2D Models.** VLM-Grounder utilizes off-the-shelf 2D open-vocabulary detectors and
88 segmentation models to filter images and generate detailed image masks for projection. Despite their
89 strengths, these 2D foundation models are not infallible. Issues like missed detections, false detections, or
90 incorrect segmentations can prevent VLM-Grounder from identifying the target object, lead to selecting the
91 wrong object, or produce noisy target masks. This noise can result in inaccurate 3D bounding box projections.

92 **Noise from Sensors.** VLM-Grounder predicts the 3D bounding box of the target object from 2D images,
93 relying on accurate camera intrinsics, extrinsics, and depth maps. However, in datasets like ScanNet [5], these
94 parameters often contain noise. For instance, depth sensors can be inaccurate at object boundaries, and RGB
95 images may suffer from motion blur. Such sensor noise leads to inaccuracies in the predicted 3D bounding
96 boxes. While sensor noise is an unavoidable challenge in robotic vision, VLM-Grounder attempts to mitigate
97 these issues through its grounding and feedback scheme combined with multi-view ensemble projection.
98 However, it cannot completely eliminate the effects of sensor inaccuracies. In practical robotic deployments,
99 robots typically have multiple types of sensors. Using multi-sensor fusion can help reduce noise and improve
100 VLM-Grounder’s performance.



Figure 1: **Demo of VLM-Grounder.**





References

- [1] T. Ren, Q. Jiang, S. Liu, Z. Zeng, W. Liu, H. Gao, H. Huang, Z. Ma, X. Jiang, Y. Chen, et al. Grounding dino 1.5: Advance the” edge” of open-set object detection, 2024.
- [2] T. Cheng, L. Song, Y. Ge, W. Liu, X. Wang, and Y. Shan. Yolo-world: Real-time open-vocabulary object detection. *arXiv preprint arXiv:2401.17270*, 2024.
- [3] G. Jocher, A. Chaurasia, and J. Qiu. Ultralytics YOLO, Jan. 2023. URL <https://github.com/ultralytics/ultralytics>.
- [4] D. Z. Chen, A. X. Chang, and M. Nießner. Scanrefer: 3d object localization in rgb-d scans using natural language. In *ECCV*, 2020.
- [5] A. Dai, A. X. Chang, M. Savva, M. Halber, T. Funkhouser, and M. Nießner. Scannet: Richly-annotated 3d reconstructions of indoor scenes. In *CVPR*, 2017.