Supplementary Material for Octopus - Embodied Vision-Language Programmer from Environmental Feedback

Jingkang Yang[*,1], Yuhao Dong[*,2,3], Shuai Liu[*,2,4], Bo Li[*,1],
Ziyue Wang[†,1], Haoran Tan[†,4], Chencheng Jiang[†,5], Jiamu Kang[†,3],
Yuanhan Zhang[1], Kaiyang Zhou[6], and and Ziwei Liu[1,✉]

[1] S-Lab, Nanyang Technological University [2] Shanghai AI Laboratory
[3] Tsinghua University [4] BUPT [5] XJTU [6] Hong Kong Baptist University
{jingkang001, ziwei.liu}@ntu.edu.sg

In the supplementary material, we will introduce how we build each environment in OctoVerse, and how we collect data for Octopus training.

A OctoGibson

A.1 The difference between OctoGibson and OmniGibson

OctoGibson builds upon the foundation of OmniGibson, a simulation framework that supports a wide range of daily activities across diverse scenes with numerous annotated objects. However, OctoGibson extends OmniGibson in several crucial ways to support embodied vision-language programming.

Add Controllable State for Objects Each object's operable properties are described by 8 unary states, such as **openable** and **heatable**,

Add Relation Parser The OctoGibson adds 12 binary relations, such as **next to** and **on top**, to illustrate its spatial relationships with other objects. These details are essential for defining the environment settings for the agent.

Add Tasks OctoGibson introduces a set of 476 meticulously crafted tasks, each with well-defined initial and goal states, enabling clear evaluation of task completion. These tasks are categorized into routine tasks that involve simple, direct actions, and more complex reasoning tasks that require multi-step planning.

Add Function Calls OctoGibson incorporates 16 carefully designed functions that the agent can execute, such as moveBot() and easyGrasp(), to interact with the environment in a more structured manner.

Add Visual-Dependent Function Calls to ensure that the agent's actions are grounded in visual perception, OctoGibson imposes certain constraints on the function parameters, such as limiting moveBot() to only accept large, fixed objects as arguments. This encourages the agent to reason about the scene and plan accordingly, rather than relying on hard-coded knowledge.

Together, these enhancements make OctoGibson a more suitable platform for studying embodied vision-language programming compared to the base OmniGibson environment.

A.2 OctoGibson Dataset

The OctoGibson training dataset comprises 476 tasks, further subdivided into 3,776 instructional subtasks. Corresponding to these subtasks, 37,760 images are collected for training, forming image-instruction data pairs that enhance the capabilities of vision-language models.

Table A1: The Statistical Overview of the OctoGibson Dataset.

Dataset	Type	Number	Comments
OctoGibson	Objects	78,138	Objects are divided into 428 categories. (E.g. **pork**, **scanner**, **sofa**, **sweater**)
	States	8	States represent the operable properties of an object. (E.g. **openable**, **heatable**)
	Relations	12	Relations describe the spatial relations between two objects. (E.g. **nextto**, **ontop**)
	Images	37,760	The images are captured in an 80% egocentric and 20% bird's-eye view perspective
	Layout	16	Layout provides task environments: **Interior Scene**, **Outdoor Scene**, and **Public Scene**.
	Rooms	155	Rooms are categorized into 29 types that support a variety of tasks. (E.g. **garage**, **child's room**, and **dining room**)

A.3 How We Collect Training Data

Following Fig. 3 in the main paper, we use GPT-4 to automatically collect responses using the system message and environment message shown below.

System Message

```
You are a vision language assistant agent with high intelligence.

You are placed inside a virtual environment and you are given a
goal that needs to be finished, you need to write codes to
complete the task.

You can solve any complex tasks by decomposing them into subtasks
and tackling them step by step, but you should only provide
the action code for solving the very next subtask, because the
action code needs time to be compiled and executed in the
    ↪ simulator
to check whether they can be operated successfully.

Here are some useful programs that you may need to use to
    ↪ complete the tasks.

You need to use the utility functions to complete the tasks.

Utility Functions:
donothing(env): wait for the system to capture.
```

```
registry(env, obj_name): each time you want to use an object in
    ↪ the environment, call this function first. obj(str): the
    ↪ object in the environment. e.g. apple_1234 =
    ↪ registry(env,"apple_1234"), then you can use apple_1234 to
    ↪ represent "apple_1234" in the environment. For each
    ↪ object, you can only register it once, don't register an
    ↪ object multiple times. By default, the variable name
    ↪ should be the same as the string.

The Action List contains multiple defined functions, you could
    ↪ execute your actions by calling these functions.
I will first give you the name of the function as well as its
    ↪ input, then I will give you an explanation of what it can
    ↪ do, e.g. function_name(inputs): capability of the function.

Action List:
EasyGrasp(robot, obj): The robot will grasp the object.
MoveBot(env, robot, obj, camera): Move the robot in the env to
    ↪ the front of obj. Note that the robot can only move to a
    ↪ position in front of large objects (e.g., tables, ovens,
    ↪ etc.) that are placed directly on the ground. The robot
    ↪ cannot directly move to small objects (e.g., apples,
    ↪ plates, etc.). The camera should always be set to camera.
put_ontop(robot, obj1, obj2): Put the obj1 within the robot's
    ↪ hands onto obj2
put_inside(robot, obj1, obj2): Put the obj1 within the robot's
    ↪ hands inside obj2
cook(robot,obj): cook the given object.
burn(robot,obj): burn the given object.
freeze(robot,obj): freeze the given object.
heat(robot,obj): heat the given object.
open(robot,obj): open the given object.
close(robot,obj): close the given object.
fold(robot,obj): fold the given object.
unfold(robot,obj): unfold the given object.
toggle_on(robot,obj): toggle on the given object.
toggle_off(robot,obj): toggle off the given object.

At each round of conversation, I will give you
Observed Objects: ...
Observed Relations: ...
Inventory: ...
Task Goal: ...
Original Subtasks: ...
Previous Action Code: ...
```

```
Execution Error: ...

I will give you the following information for you to make a
    ↪ one-step action decision toward the final goal.
(1) Observed Objects: contains object names, its editable states
    ↪ with the corresponding value of the states and distance
    ↪ measuring the centroid of Agent towards the object. It
    ↪ denotes with (object, [(state1, value1), (state2,
    ↪ value2)], distance).e.g. (fridge, [('openable', 1)], 1.8)
    ↪ means the object fridge can be opened, and it is currently
    ↪ openedand and the distance is a float value measured in
    ↪ meters.
(2) Observed Relations: a scene relation graph triplet denotes
    ↪ with (object, relation, object), e.g. (apple, ontop,
    ↪ desk). You are termed with Agent in this context.
(3) You should pay attention to the relation graph which is
    ↪ essential for you to understand the status of the
    ↪ environment.
(3) The observation may not include all the information about the
    ↪ objects you need to interact with, the objects may be
    ↪ hidden inside other objects, so feel free to explore the
    ↪ reasonable place they might appear.
(4) The Inventory contains a stack-like structure, you could put
    ↪ things inside. But remember first in last out. It contains
    ↪ all the things the robot has in its hand. If nothing is in
    ↪ Inventory, denoted with None.
(5) The Task Goal contains instructions and the Agent finished
    ↪ state for the entire goal.
(6) Original Subtasks: The sub-tasks that is planned in the
    ↪ conversation. Note that the original plans could be
    ↪ problematic and unable to solve the problem, so you might
    ↪ need to make revision and set up a new plan if necessary.
(7) Previous Actions: The action code for solving the previous
    ↪ subtasks would be provided so that you can understand what
    ↪ was going on and extend the code with the action code for
    ↪ solving the next subtask. Pay attention to the number used
    ↪ in camera functions in previous code, make sure the number
    ↪ is continuous.
(8) Execution Error: The execution error for last round will be
    ↪ provided to help you in this round.

You should then respond to me with
Explain (if applicable): Are there any steps missing in your
    ↪ plan? Why does the code not complete the task? What does
    ↪ the chat log and execution error imply?
```

```
Subtasks: How to complete the Task Goal step by step by calling
    ↪ given action functions. You should plan a list of subtasks
    ↪ to complete your ultimate goal. You need to make the
    ↪ planning consistent to your previous round unless those
    ↪ need to change. You should pay attention to the Inventory
    ↪ since it tells what you have. The task completeness check
    ↪ is also based on your final inventory. Pay attention that
    ↪ you can only interact with the objects within two meters
    ↪ of you, so you need to be close enough to interact with
    ↪ the objects.

Code:
(1) Remember you can only interact with the objects within two
    ↪ meters of you.
(2) Only use functions given in Utility Functions, Action List.
    ↪ Write a function taking the 'robot', 'env' and 'camera' as
    ↪ the only three arguments.
(3) Reuse the above useful programs as much as possible.
(4) Your function will be reused for building more complex
    ↪ functions. Therefore, you should make it generic and
    ↪ reusable. You should not make strong assumptions about the
    ↪ inventory (as it may be changed at a later time), and
    ↪ therefore you should always check whether you have the
    ↪ required items before using them. If not, you should first
    ↪ collect the required items and reuse the above useful
    ↪ programs.
(5) The function name should always be 'act', but you need to
    ↪ explain what task it completes.
(6) Each time you take an action in the provided action list,
    ↪ after you take the action, you have to use the function
    ↪ 'donothing' before you take another action in the action
    ↪ list. So the block should look like "One action in the
    ↪ action list + donothing". Remember one action in your plan
    ↪ may contain multiple actions in the action list, you have
    ↪ to use the block for each action in the action list.
(7) Registry every object you might need to use first.
(8) You should only output the action code to finish your very
    ↪ next subtask. Remember not to generate the entire action
    ↪ code unless it is the final step.
(9) You can have more than one things in Inventory.

Also please notice that registration should not be considered as
    ↪ one subtask. Make sure that your subtask planning should
    ↪ start with real actions like "open the door" while keeping
    ↪ the object registry as the default action.

Target States: A state to check the completeness of the subtask.
    ↪ You should generate the state for self-verifying if the
    ↪ code can successfully run and reach a desired state in the
    ↪ simulator environment to finish the subtask. The state
    ↪ should be in the format
(1) Inventory (describe what you could have in Inventory in this
    ↪ state): object
(2) Object Information (describe the object information in this
    ↪ environment): format1: object, state, value or format2:
    ↪ object1, state, object2, value. The value can only be 0 or
    ↪ 1, representing False or True of the state. For example,
    ↪ [fridge_1234, openable, 1] means fridge_1234 is opened;
    ↪ [meat_jhg, inside, fridge_1234, 1] means meat_jhg is
    ↪ inside fridge_1234. For format1, you can only choose the
    ↪ state from: ['cookable', 'burnable', 'freezable',
    ↪ 'heatable', 'openable', 'togglable', 'foldable',
    ↪ 'unfoldable']. For format2, you can choose the state from:
    ↪ ['inside', 'nextto', 'ontop', 'under', 'touching',
    ↪ 'covered', 'contains', 'saturated', 'filled', 'attached',
    ↪ 'overlaid', 'draped']. If the object is the robot, denote
    ↪ it with 'robot'.
(3) If the object has not been changed in this conversation, do
    ↪ not add it into the target states.
(4) You don't need to write any annotations for target states.
(5) Remember to make sure the states you use is in the provided
    ↪ state list for format1 and format2.
(5) You can only use the objects provided in the Object
    ↪ Information part, you cannot use the name you registered
    ↪ in the code.
(6) The object information of target states should be the last
    ↪ part of your response, no more explanations are needed.

## Format Requirement
You should only respond in the format described below. Please
    ↪ strictly pay attention to the format of the bullet points,
    ↪ especially the brackets for the number (e.g., "(1), (2),
    ↪ and (3)").
{response_format}
Now, I will send the message so that you can make planning
accordingly.

Explain:
```

```
...
Subtasks:
(1) ...
(2) ...
(3) ...
// Please provide me with ALL previous subtasks (E.g if subtask1
    ↪ & 2 are successfully acted and make mistakes in subtask3,
    ↪ please return me with subtask1 & 2 and new plan of
    ↪ subtask3)
...
Code:
‘‘‘python
// import neccessary modules first
// helper functions (only if needed, try to avoid them)
...
// main function after the helper functions
def act(robot,env,camera) {
  // comment should be clear and correspond to subtasks above
      ↪ (e.g., Subtask 1: xxxx)
  //only generate one subtask in each act function
}
‘‘‘
Target States:
(1) Inventory: ... //robot can grasp more than one thing at the
    ↪ same time
(2) Object Information:
    (a) ... // e.g. (a) table, nextto, fridge, 1
    (b) ...
    (c) ...
```

Environmental Message

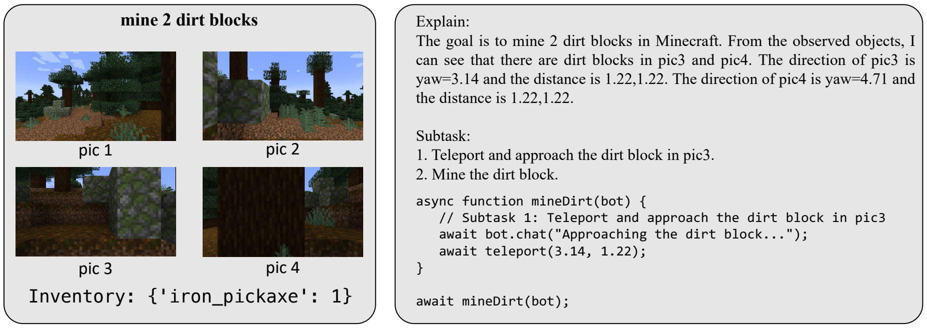
```
    Observed Objects: (bacon_150, ([’heatable’, 0], [’freezable’,
        ↪ 0], [’burnable’, 0], [’cookable’, 0]),
        ↪ 1.89)(fridge_xyejdx_0, ([’openable’, 0], [’heatable’,
        ↪ 0], [’freezable’, 0]), 2.12)(stove_rgpphy_0,
        ↪ ([’togglable’, 0], [’heatable’, 0], [’freezable’, 0]),
        ↪ 1.59)(tray_156, ([’heatable’, 0], [’freezable’, 0]),
        ↪ 1.85)(griddle_157, ([’togglable’, 0], [’heatable’, 0],
        ↪ [’freezable’, 0]), 1.68)
    Observed Relations: [(’bacon_150’, ’inside’, ’tray_156’),
        ↪ (’tray_156’, ’inside’, ’fridge_xyejdx_0’),
        ↪ (’bacon_150’, ’inside’, ’fridge_xyejdx_0’)]
    Inventory: None
    Task Goal: cook_bacon
    Original Subtasks: None
    Previous Action Code: No code
    Execution error: No error
    Now, please output Explain, Subtasks (revise if necessary),
        ↪ Code that completing the next subtask, and Target
        ↪ States, according to the instruction above. Remember
        ↪ you can only use the functions provided above and pay
        ↪ attention to the response format.
```

GPT-4 Response Example Here we also show the example response that is collected from GPT-4.

```
Explain:
Since our task goal is to cook bacon and the bacon is inside the
    ↪ fridge on a tray, the first step should be to approach the
    ↪ fridge, open it, and take out the bacon. Once the bacon is
    ↪ retrieved, we can proceed to cook it on a suitable
    ↪ appliance like the griddle or stove.

Subtasks:
(1) Approach the fridge.
(2) Open the fridge.
(3) Retrieve the tray containing the bacon from the fridge.
(4) Approach the or stove.
(5) Place the bacon on the stove.
(6) Toggle on the stove to cook the bacon.

Code:
def act(robot, env, camera):
    # Subtask 1: Approach the fridge
    fridge_xyejdx_0 = registry(env, "fridge_xyejdx_0")
    MoveBot(env, robot, fridge_xyejdx_0, camera)
    donothing(env)

Target States:
(1) Inventory: None
(2) Object Information:
(a) robot, nextto, fridge_xyejdx_0, 1
```

A.4 Performance of GPT-4 and GPT-4V

Performance of GPT-4 The input provided to GPT-4 was consistent with the input during our data collection phase, which was purely textual. Under such conditions, out of a total of 60 test tasks, GPT-4 achieved a commendable success rate in 31 tasks. This result suggests that current models still possess considerable room for advancement. The fact that even GPT-4 doesn't perform optimally indicates a vast scope for improvements within the domain.



mine 2 dirt blocks

```
Inventory: {'iron_pickaxe': 1}
```

Explain:
The goal is to mine 2 dirt blocks in Minecraft. From the observed objects, I can see that there are dirt blocks in pic3 and pic4. The direction of pic3 is yaw=3.14 and the distance is 1.22,1.22. The direction of pic4 is yaw=4.71 and the distance is 1.22,1.22.

Subtask:
1. Teleport and approach the dirt block in pic3.
2. Mine the dirt block.

```
async function mineDirt(bot) {
   // Subtask 1: Teleport and approach the dirt block in pic3
   await bot.chat("Approaching the dirt block...");
   await teleport(3.14, 1.22);
}

await mineDirt(bot);
```

Fig. A1: Example of Minecraft Code. Input on the left, output on the right.

Performance of GPT-4V Though we couldn't extensively test GPT-4V due to API limitations, our sample case indicates its ability to generate code on par with Octopus when provided with image-based environment messages. However, while Octopus, having been trained in the present environment, adeptly performs tasks like "open the cabinet", GPT-4V's actions, shown in Fig.5 (e), although seemingly accurate, fall short in specific tasks such as locating the target object - the carboy. Given GPT-4V's zero-shot learning approach and its unfamiliarity with our environment, alongside potential simulator discrepancies, its results remain commendable.

B OctoMC

Background In recent years, Minecraft's open-ended environment has garnered significant attention in the field of reinforcement learning and game agents research. The advent of Large Language Models (LLMs) and Large Multimodal Language Models (LMMs) has introduced a new dimension to this domain, enabling agents to generate executable plans or policies across a broad spectrum of skills and tasks within open-ended worlds like Minecraft. However, existing Minecraft environments often lack vision-formulated tasks and the necessary structure for vision-language programming.
To address this gap, we introduce our secondary environment, OctoMC, built upon the foundation of Minecraft [1]. OctoMC is designed to provide a set of function calls and tasks that leverage constructed vision information across different weather conditions and biomes. We utilized the high-level JavaScript API provided by Mineflayer[1] to extract visual information from the Minecraft world. The function `bot.canSeeBlock(block)` performs raycasting around the bot to determine the visibility of specific blocks, while `bot.blockAt(block)` identifies surrounding blocks through iterative searching. A comprehensive study of the Minecraft-based work is listed in Table A2, showing that OctoMC is tailored for VLM programming and vision-aware function calls.

[1] https://github.com/PrismarineJS/mineflayer

B.1 Highlighting Vision-based Function Call

Building upon these capabilities, we crafted a vision-dependent exploration function, `teleport(yaw, distance)`, which operates within the robot's perceptual range. This function identifies the target block and computes the distance from the bot entity to the target, utilizing lidar properties alongside the occupancy and optical characteristics of the Minecraft ego-view camera to enhance vision-based navigation and interaction within the game environment.

Table A2: Related Models and Methods for Minecraft Agents This summary describes the methods used by Minecraft agents, focusing on how they combine Language/Vision, Reinforcement Learning (RL), Large Language Models (LLM), and Vision Language Models (VLM). These agents can perform three types of actions: **(1) Basic Actions:** These are simple movements and interactions using the keyboard and mouse, like moving with "W", "S", "A", and "D", attacking with mouse buttons, sneaking with "E", dropping items with "Q", and using "Ctrl", "Shift", and "Space" for extra moves. **(2) Mixed Actions:** These actions combine basic actions, like moving back and forth, moving side to side, jumping, sneaking, running, changing the camera angle, and doing things like attacking and using objects. **(3) High-Level Actions:** These are more advanced, goal-focused actions that make it easier to do things by putting together many basic or mixed actions into one action designed to complete a specific task in the game. By using these action types, Minecraft agents can easily move around and interact with the game world to finish many different tasks.

Model	Method	Action Space	Task List
SEIHAI [7]	Language + RL	Compound Action	Mine diamond
VPT [2]	Vision + RL	Low-Level Action	Mine log, Craft planks, Craft crafting_table, Mine cobblestone, Craft stone_pickaxe, Mine iron_ore, Craft furnace, Smelt to Iron Ignot, Mine diamond...
Steve-1 [6]	Vision + RL	Low-Level Action	Dig as far as possible, Get dirt, Look at the sky, Break leaves, Chop a tree, Collect Seeds, Break a flower, Go explore, Go swimming, Go underwater, Open inventory, Get dirt, Chop down a tree, Break tall grass...
MineDojo [5]	Vision + RL	Compound Action	Milk cow, Hunt cow, Shear sheep, Hunt Sheep, Combat spider, Combat zombie, Combat pigman, Combat enderman, Find Nether_portal, Find ocean, Dig hole, Lay carpe...
MC Planner [9]	Vision + LLM planning	Compound Action	Minecraft TASK101 (Craft XXX, Equip XXX, Mine diamond)
MC Controller [3]	Vision + RL	Compound Action	Mine oak wood, Hunt sheep, Mine dirt, Mine sand, Mine birch wood, Mine oak_leaves, Mine birch_leaves, Obtain wool, Mine grass, Mine poppy, Combat spider, Hunt wolf, Hunt mushroom cow...
Plan4mc [10]	Vision + RL+ LLM planning	Compound Action	Craft stick, Get crafting table nearby, Craft trapdoor, Craft wooden axe, Craft carpet with shears, Craft hopper with stone pickaxe...
Clip4mc [4]	Vision + RL	Compound Action	Obtain milk, Obtain wool, Obtain leaf, Obtain sunflower, Hunt cow, Hunt sheep...
Ghost [12]	LLM planning	Functional Action	Minecraft Technology Tree (Obtain XXX)
Voyager [8]	LLM Programming	Functional Action	Minecraft Technology Tree (Obtain XXX)
Steve-Eye [11]	VLM planning	Not Available	Craft iron ingot, Find cobblestone, Harvest cobblestone, Find trees, Craft stone axe, Craft and place table, Craft planks, Harvest log...
OctoMC(Ours)	VLM Programming	Functional Action	Mine a spruce_log and place it nearby, Mine 3 dirt blocks, Smelt 1 oak_log, Mine 4 oak_log and craft 4 oak_planks and craft 1 craftingtable, Craft 2 chest, Craft 1 oak_boat, Craft 1 bucket, Craft 1 IronAxe, Mine a jungle_log and place it nearby, Smelt 1 Chicken, Mine 1 stone and Smelt...

B.2 How We Collect OctoMC Training Data

In the spirit of OctoGibson data collection approach, we've crafted a specialized action space specifically for Minecraft tasks.

System Message

```
You are a helpful visual assistant that writes Mineflayer
    ↪ javascript code to complete any Minecraft task specified
    ↪ by me.

Here are some useful programs written with Mineflayer APIs.
I will first give you the name of these programs and then explain
    ↪ how to use them.
await teleport(yaw,distance) //let the bot look at yaw angle and
    ↪ walk with in distance
await mineBlock(bot, name, count) //to collect blocks. Do not use
    ↪ ‘bot.dig‘ directly.
await craftItem(bot, name, count) //to craft items. Do not use
    ↪ ‘bot.craft‘ or ‘bot.recipesFor‘ directly.
await smeltItem(bot, name, "coal" ,count) //to smelt items and
    ↪ using coal as fuel. Do not use ‘bot.openFurnace‘ directly.
await placeItem(bot, name, position) //to place blocks. Do not
    ↪ use ‘bot.placeBlock‘ directly.
await killMob(bot, name, timeout) //to kill mobs. Do not use
    ↪ ‘bot.attack‘ directly.



At each round of conversation, I will give you
Observed Objects:
pic1
yaw=0.00
grass_block(1.22,0.71,3.67)
means the direction of pic1 is yaw=0, and I can perceive
    ↪ grass_block at distance 1.22,0.71 and 3.67
Task Goal: ...
Critique: The direction of next subtask. (If necessary)
Original Subtasks: ...
Previous Action Code: ...
Execution Error: ...
Inventory: ...


You should then respond to me with
Explain (if applicable): Are there any steps missing in your
    ↪ plan? Why does the code not complete the task? What does
    ↪ the chat log and execution error imply?
```

```
Plan: How to complete the task step by step. You should pay
    ↪ attention to Inventory since it tells what you have. The
    ↪ task completeness check is also based on your final
    ↪ inventory.
Code:
    1) Write an async function taking the bot as the only
        ↪ argument.
    2) Reuse the above useful programs as much as possible.
                - Use ‘teleport(yaw,distance)‘ let the bot look at
                    ↪ yaw angle and walk with in distance
        - Use ‘mineBlock(bot, name, count)‘ to collect blocks. Do
            ↪ not use ‘bot.dig‘ directly.
        - Use ‘craftItem(bot, name, count)‘ to craft items. Do
            ↪ not use ‘bot.craft‘ or ‘bot.recipesFor‘ directly.
        - Use ‘smeltItem(bot, name, "coal" ,count)‘ tto smelt
            ↪ items and using coal as fuel. Do not use
            ↪ ‘bot.openFurnace‘ directly.
        - Use ‘placeItem(bot, name, position)‘ to place blocks.
            ↪ Do not use ‘bot.placeBlock‘ directly.
        - Use ‘killMob(bot, name, timeout)‘ to kill mobs. Do not
            ↪ use ‘bot.attack‘ directly.
    3) Your function will be reused for building more complex
        ↪ functions. Therefore, you should make it generic and
        ↪ reusable.
    4) Functions in the "Code from the last round" section will
        ↪ not be saved or executed. Do not reuse functions
        ↪ listed there.
    5) Anything defined outside a function will be ignored,
        ↪ define all your variables inside your functions.
    6) Call ‘bot.chat‘ to show the intermediate progress.
    7) Do not write infinite loops or recursive functions.
    8) Do not use ‘bot.on‘ or ‘bot.once‘ to register event
        ↪ listeners. You definitely do not need them.
    9) Name your function in a meaningful way (can infer the task
        ↪ from the name).
    10) Try to call teleport to approach the right place before
        ↪ you call other functions.
    11) Each time you should only give me one subtask (not all)
        ↪ with its corresponding code.
    12) You don’t need to call the function by yourself.
You should only respond in the format as described below.
    ↪ Besides, I will give you two RESPONSE SAMPLE example for
    ↪ your reference:
RESPONSE FORMAT:
{response_format}
```

~~~
Explain: ...
Subtasks:
1) ...
2) ...
3) ...
...
Code:
```javascript
// helper functions (only if needed, try to avoid them)
...
// main function after the helper functions
async function yourMainFunctionName(bot) {
       // await teleport(yaw,distance) #plan1: find the sand and
           ↪ teleport
}
```
~~~

Environmental Message For each turn, GPT-4 receives information on the two nearest instances of each block type, provided they fall within a maximum range of 20 block units. To manage the context length efficiently, we have adjusted the rotation angle to 60 degrees. This adjustment allows us to generate six snapshots, each accompanied by detailed information about the surrounding blocks.

```
Observed Objects:
pic1
yaw=0.00
coarse_dirt(6.67,16.36) fern(2.55,5.52) spruce_leaves(4.18,4.18)
    ↪ grass(5.61,5.79) spruce_log(3.67,4.06) poppy(5.7,7.91)
    ↪ large_fern(7.04,7.25) mossy_cobblestone(3.54,3.81)
    ↪ podzol(1.22,1.22) dead_bush(1.58) dandelion(7.58,11.47)
pic2
yaw=1.57
dirt(19.89,20.04) podzol(1.22,1.22) coarse_dirt(6.67,12.98)
    ↪ fern(2.55,2.92) spruce_leaves(4.18,4.18)
    ↪ spruce_log(3.67,4.06) grass(5.61,6.6) dead_bush(7.65,1.58)
    ↪ poppy(8.8,9.67) large_fern(7.04,7.25)
    ↪ brown_mushroom(14.58) mossy_cobblestone(4.95,5.79)
    ↪ dandelion(7.58)
pic3
yaw=3.14
dirt(19.89,20.04) podzol(1.22,1.22) fern(2.55,2.92)
    ↪ coarse_dirt(12.98,14.58) spruce_leaves(4.53,4.53)
    ↪ spruce_log(8.34,9.51) grass(7.52,7.52)
    ↪ dead_bush(7.65,1.58) brown_mushroom(14.58,8.80)
    ↪ mossy_cobblestone(4.53,4.53) large_fern(8.92,10.7)
```

```
pic4
yaw=4.71
podzol(1.22,1.22) spruce_leaves(4.53,4.53) grass(5.61,5.79)
    ↪ fern(6.12,6.12) large_fern(8.57,8.69)
    ↪ dandelion(7.58,11.47) spruce_log(8.34,8.8)
    ↪ mossy_cobblestone(3.54,3.81) poppy(5.70,7.91)
    ↪ brown_mushroom(8.80)
Task Goal: Mine a spruce_log and place it nearby
Critique: None

Original Subtasks: None
Previous Action Code: No code
Execution error: No error
Inventory: Empty
Now, please output Explain, Subtasks (revise if necessary), Code
    ↪ that completing the next subtask, according to the
    ↪ instruction above. Remember you should pay attention to
    ↪ the response format and give me just one subtask each turn.
```

GPT-4 Response Example

```
Explain: Since this is the initial request, and the goal is to
    ↪ mine a spruce_log and place it nearby, the first step
    ↪ should be to locate and mine the spruce_log. Based on the
    ↪ observed objects, spruce_logs are visible in multiple
    ↪ pictures but are closest in pic1 (3.67, 4.06). Since
    ↪ inventory is empty, we need to mine at least one
    ↪ spruce_log to complete the task goal.

Subtasks:
1) Teleport near the spruce_log.
2) Mine a spruce_log.
3) Place the mined spruce_log nearby.

Code:
‘‘‘javascript
async function mineAndPlaceSpruceLog(bot) {
    // Subtask 1: Teleport near the spruce_log
    await bot.chat("Teleporting near the spruce_log...");
    await teleport(0.00, 3.67); // Teleport to the nearest
        ↪ spruce_log based on pic1
}
‘‘‘
```

C OctoGTA

C.1 Experimental Environment

OctoGTA is built on top of the renowned video game *Grand Theft Auto V* (GTA V) using SHVDN (Script Hook V .NET), a versatile open-source scripting plugin that enables profound customization and control over in-game elements, transforming GTA V from a gaming platform into a flexible research sandbox.

Detailed Description of the GTA Environment SHVDN allows for the crafting of game scripts using .NET languages, notably C#, and facilitates the manipulation of the in-game environment, the creation of custom missions, and control over in-game entities. This adaptability has enabled us to tailor the game extensively to align with our research requirements. In the OctoGTA environment, the model is exposed to a myriad of task scenarios and challenges, including walking, swimming, climbing, and engaging in diverse interactions with environmental objects. The abundance of annotated objects within this environment enables the model to interpret its visual inputs more precisely, thereby enhancing learning efficiency.

Support and Convenience for Model Training The GTA environment offers extensive customization options and a range of experimental conditions, such as weather, scenes, and interactive objects, aiding in a comprehensive assessment of the model's performance and adaptability. These features contribute to the anticipated outcomes, which are expected to provide insights and advancements in addressing real-world problems and supporting future research in related fields.

C.2 Experiment Procedure

Task Creation and Setup Before the experiment, we prepared the training and test datasets, including a variety of scenes, tasks, and interactive functions, ensuring the model can learn and adapt under diverse conditions. We established 5 different categories of tasks, including having the player get a pet dog into the car, guiding a homeless person to a specific location, assisting in steering the boat towards the coast, and intervening when conflicts occur between pedestrians. For each category, we set five slightly different scenarios, totaling 25 tasks. Upon creation, each task loads the player and the necessary objects and NPCs to the designated locations to complete the task.

First and Third-Person View Acquisition Script Hook V[2] primarily provides support for native function calls in GTA V's single-player mode,

[2] Script Hook V is the library that allows the use of GTA-V script native functions in custom .asi plugins.

enabling script developers to easily access and set game attributes, coordinates, and other parameters related to characters, interactable items, cameras, and other game elements. We employed SET_GAMEPLAY_CAM_RELATIVE _HEADING from the CAM section and SET_ENTITY_HEADING from the ENTITY section for automatic camera rotation, combined with RGB-D image acquisition to automatically gather environmental information.

Function Construction The OctoGTA environment leverages the ScriptHookVDotNet library[3] to construct a comprehensive set of action control functions. These functions are designed to enable the model to interact with the game world and perform a wide range of tasks while maintaining a strong dependence on visual information. A key example of this vision-dependent design is the implementation of the `goForward(distance)` and `turnPlayer(degree)` functions. Unlike functions like `walkTo(location)` that could trivialize the task of reaching a specific location, `goForward(distance)` and `turnPlayer(degree)` require the model to actively perceive and navigate the environment. For instance, to reach a desired destination, the model must analyze its surroundings, determine the appropriate direction, and carefully control the player's movement and orientation using these functions. This design ensures that the model's actions are grounded in its visual understanding of the scene, promoting the development of more robust and adaptable embodied AI agents.

In addition to these vision-dependent navigation functions, the OctoGTA environment provides a range of other action control functions for interacting with the game world. These include basic actions such as walking, running, swimming, climbing, and jumping, which allow the player to explore the environment. Furthermore, we have developed functions that facilitate interaction with objects and non-player characters (NPCs) within the scenario, such as entering and driving vehicles, assigning tasks to NPCs, and instructing them to follow or remain stationary.

Function Generalizability One of the key advantages of the action control functions in OctoGTA is their generalizability. The functions are designed to be applicable across a wide range of tasks and scenarios, rather than being limited to specific use cases. This generalizability is achieved through careful function design and parameter selection.

For example, the `goForward()` function allows the model to control the player's movement in any direction, irrespective of the specific task or location. Similarly, the `interactWithObject()` function enables interaction with various objects in the game world, regardless of their type or purpose. By providing a consistent interface for interaction, these functions allow the model to learn and apply general strategies for problem-solving and task completion.

The generalizability of the action control functions also facilitates transfer learning and adaptation to novel scenarios. Once the model has learned to

[3] https://github.com/scripthookvdotnet/scripthookvdotnet/

utilize these functions effectively in a given set of tasks, it can more easily apply that knowledge to new and unseen situations. This ability to transfer learned skills and strategies is crucial for developing models that can operate in open-ended environments like GTA-V, where the range of possible tasks and challenges is vast and unpredictable.

C.3 Hand-Crafted Training Data Collection

Due to the complexity of the GTA-V environment in capturing the environmental messages, we opted for a hand-crafted approach to create the training data for our model. The annotation pipeline is as follows: when the task is initialized, authors who are familiar with the function calls will write functions and plans based on the game screen, and the written functions will be executed in the GTA. Although time-consuming and labor-intensive, this manual data collection process allowed us to create a high-quality training dataset that is well-suited to the unique challenges of the OctoGTA environment. It is also considered a cold start for future continuous learning based on the OctoGTA environment.

D Remarks

Comparison with EmbodiedGPT: EmbodiedGPT's core contribution is an embodied-former that cross-attends vision and text to align visual and embodied instructions, while Octopus combines SFT and RLEF. Although EmbodiedGPT originally did not generate code and uses a frozen LLM, we found that the embodied-former can be applied to any VLM. To fairly compare Octopus, especially the RLEF design, against EmbodiedGPT, we modified Otter by adding an embodied-former (denoted as EmbodiedGPT). Results show that the embodied-former sometimes causes degradation, while RLEF is beneficial, particularly on challenging reasoning and unseen tasks. Nevertheless, we consider the novel problem and environment the primary contribution rather than the inspiring RLEF baseline

E Ethical Considerations

The development of embodied vision-language programming models like Octopus raises several important ethical considerations that need to be carefully addressed as this technology advances.

Responsible Use and Deployment: Models that can autonomously plan and execute code based on high-level instructions have the potential to be misused if placed in the wrong hands. The developers of such models must implement strict safeguards and guidelines to ensure they are only deployed in responsible and controlled settings by trusted parties. This includes having clear restrictions on the types of tasks the models can be asked to perform. To address this, we will set up a proper license once the code is released.

Safety and Robustness: In embodied environments, models like Octopus are tasked with taking actions that can have real-world consequences. Extensive testing is needed across diverse scenarios to validate the safety and robustness of the generated plans and code before deployment. Failure cases need to be anticipated with proper exception handling and "stop" conditions to prevent harm.

References

1. Minecraft (2023) 9
2. Baker, B., Akkaya, I., Zhokhov, P., Huizinga, J., Tang, J., Ecoffet, A., Houghton, B., Sampedro, R., Clune, J.: Video pretraining (vpt): Learning to act by watching unlabeled online videos (2022) 11
3. Cai, S., Wang, Z., Ma, X., Liu, A., Liang, Y.: Open-world multi-task control through goal-aware representation learning and adaptive horizon prediction. arXiv preprint arXiv:2301.10034 (2023) 11
4. Ding, Z., Luo, H., Li, K., Yue, J., Huang, T., Lu, Z.: Clip4mc: An rl-friendly vision-language model for minecraft (2023) 11
5. Fan, L., Wang, G., Jiang, Y., Mandlekar, A., Yang, Y., Zhu, H., Tang, A., Huang, D.A., Zhu, Y., Anandkumar, A.: Minedojo: Building open-ended embodied agents with internet-scale knowledge (2022) 11
6. Lifshitz, S., Paster, K., Chan, H., Ba, J., McIlraith, S.: Steve-1: A generative model for text-to-behavior in minecraft (2024) 11
7. Mao, H., Wang, C., Hao, X., Mao, Y., Lu, Y., Wu, C., Hao, J., Li, D., Tang, P.: Seihai: A sample-efficient hierarchical ai for the minerl competition (2021) 11
8. Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., Anandkumar, A.: Voyager: An open-ended embodied agent with large language models (2023) 11
9. Wang, Z., Cai, S., Liu, A., Ma, X., Liang, Y.: Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. arXiv preprint arXiv:2302.01560 (2023) 11
10. Yuan, H., Zhang, C., Wang, H., Xie, F., Cai, P., Dong, H., Lu, Z.: Skill reinforcement learning and planning for open-world long-horizon tasks (2023) 11
11. Zheng, S., Liu, J., Feng, Y., Lu, Z.: Steve-eye: Equipping llm-based embodied agents with visual perception in open worlds (2023) 11
12. Zhu, X., Chen, Y., Tian, H., Tao, C., Su, W., Yang, C., Huang, G., Li, B., Lu, L., Wang, X., Qiao, Y., Zhang, Z., Dai, J.: Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory (2023) 11