Multimodal Visual Understanding Applications

Note: Due to performance limitations, Raspberry Pi 5 2GB/4GB, RDK X5 4GB, and Jetson Orin Nano 4GB versions cannot run offline. Please refer to the online large model tutorial.

1. Concept Introduction

1.1 What is "Visual Understanding"?

In the `largemodel` project, the **multimodal visual understanding** function refers to enabling robots to not only "see" a pixel matrix, but also to truly "understand" the content, objects, scenes, and relationships within an image. This is like giving the robot a pair of thinking eyes.

The core tool for this function is `seewhat`. When a user gives a command such as "Look at what's here," the system calls this tool, triggering a series of background operations, ultimately feeding back the AI's analysis of the real-time image to the user in natural language.

1.2 Implementation Principle Overview

The basic principle is to input two different types of information—**images (visual information)** and **text (linguistic information)**—into a powerful multimodal large model (e.g., LLaVA).

1. **Image Encoding**: The model first uses a vision encoder to convert the input image into computer-understandable digital vectors. These vectors capture features such as color, shape, and texture of the image.
2. **Text Encoding**: Simultaneously, the user's question (e.g., "What's on the table?") is also converted into a text vector.
3. **Cross-Modal Fusion**: The most crucial step is that the model fuses the image vectors and text vectors in a special "attention layer." Here, the model learns to "pay attention" to the

parts of the image that are relevant to the question. For example, when asked about "table," the model will pay more attention to regions in the image that match the features of "table."
4. **Answer Generation**: Finally, a large language model (LLM) part generates a descriptive text as the answer based on the fused information.

In simple terms, it involves **using text to "light up" corresponding parts of an image, and then using language to describe the "lit" parts.**

---

2. Code Analysis

Key Code

1. Utility Layer Entry Point (`largemodel/utils/tools_manager.py`)

The `seewhat` function in this file defines the execution flow of this tool.

```
# From largemodel/utils/tools_manager.py

class ToolsManager:
    # ...

    def seewhat(self):
        """
        Capture camera frame and analyze environment with AI model.


        :return: Dictionary with scene description and image path, or None if
failed.
        """
        self.node.get_logger().info("Executing seewhat() tool")
        image_path = self.capture_frame()
        if image_path:
            # Use isolated context for image analysis.
            analysis_text = self._get_actual_scene_description(image_path)

            # Return structured data for the tool chain.
            return {
                "description": analysis_text,
                "image_path": image_path
            }
        else:
            # ... (Error handling)
            return None

    def _get_actual_scene_description(self, image_path, message_context=None):
        """
        Get AI-generated scene description for captured image.

        :param image_path: Path to captured image file.
        :return: Plain text description of scene.
        """
        try:
            # ... (Building Prompt)
            result = self.node.model_client.infer_with_image(image_path,
scene_prompt, message=simple_context)
```

```
            # ... (Processing results)
            return description
        except Exception as e:
            # ...
```

2. Model Interface Layer (`largemodel/utils/large_model_interface.py`)

The `infer_with_image` function in this file is the unified entry point for all image understanding tasks; it is responsible for calling the specific model implementation based on the configuration.

```python
# From largemodel/utils/large_model_interface.py

class model_interface:
    # ...
    def infer_with_image(self, image_path, text=None, message=None):
        """Unified image inference interface. """
        # ... (Prepare Message)
        try:
            # Determine which specific implementation to call based on the value
of self.llm_platform
            if self.llm_platform == 'ollama':
                response_content = self.ollama_infer(self.messages,
image_path=image_path)
            elif self.llm_platform == 'tongyi':
                # ... Logic for calling the Tongyi model
                pass
            # ... (Logic of other platforms)
        # ...
        return {'response': response_content, 'messages': self.messages.copy()}
```

Code Analysis

The implementation of this functionality involves two main layers: the tool layer defines the business logic, and the model interface layer is responsible for communicating with the large language model. This layered design is key to achieving platform versatility.

1. **Tool Layer (`tools_manager.py`):**

- The `seewhat` function is the core of the visual understanding functionality. It encapsulates the complete process of the "seeing" action: first, it calls the `capture_frame` method to obtain the image, and then calls `_get_actual_scene_description` to prepare a prompt for requesting the model to analyze the image.
- The most crucial step is that it calls the `infer_with_image` method of the model interface layer. It doesn't care which model is used underneath; it only handles passing the two core data: the "image" and the "analysis prompt."
- Finally, it packages the analysis results (plain text descriptions) received from the model interface layer into a structured dictionary and returns it. This allows upper-layer applications to easily use the analysis results.

2. **Model Interface Layer (`large_model_interface.py`):**

- The `infer_with_image` function acts as a "dispatch center." Its main responsibility is to check the current platform configuration (`self.llm_platform`) and distribute the task to the corresponding specific processing function (e.g., `ollama_infer` or `tongyi_infer`) based on the configuration values.

- This layer is crucial for adapting to different AI platforms. All platform-specific operations (such as data encoding, API call formats, etc.) are encapsulated in their respective processing functions.
- In this way, the business logic code in `tools_manager.py` can support multiple different backend large model services without any modifications. It only needs to interact with the unified and stable interface `infer_with_image`.

In summary, the execution flow of the `seewhat` tool embodies a clear separation of responsibilities: `ToolsManager` is responsible for defining "what to do" (acquiring images and requesting analysis), while `model_interface` is responsible for defining "how to do it" (selecting the appropriate model platform based on the current configuration and interacting with it). This makes the tutorial's explanation universal; the core code logic remains consistent regardless of whether the user is online or offline. 3. Practical Operation

3.1 Configuring the Offline Large Model

3.1.1 Configuring the LLM Platform (`yahboom.yaml`)

This file determines which large model platform the `model_service` node loads as its primary language model.

For Raspberry Pi 5, entering the Docker container is required; for RDK X5 and Orin mainframes, it is not necessary:

```
./ros2_docker.sh
```

If you need to enter the same Docker container to input other commands later, simply enter `./ros2_docker.sh` again in the host machine's terminal.

1. **Open the file in the terminal**:

```
vim ~/yahboom_ws/src/largemodel/config/yahboom.yaml
```

2. **Modify/confirm** `llm_platform`:

```
model_service:                                  #Model server node parameters
  ros__parameters:
    language: 'zh'                              #Large Model Interface Language
    useolinetts: True                           #This item is invalid in text mode and
can be ignored

    # Large model configuration
    llm_platform: 'ollama'                      # KEY: Make sure this is 'ollama'
    regional_setting : "China"
    camera_type: 'csi'                         # Camera type: 'csi', 'usb'
    mjpeg_stream_url: 'http://172.17.0.1:8080/camera.mjpg' # MJPEG stream URL
for CSI camera in Docker
```

3.1.2 Configuring the Model Interface (`large_model_interface.yaml`)

This file defines which visual model to use when the platform is selected as `ollama`.

1. Open the file in the terminal

```
vim ~/yahboom_ws/src/largemodel/config/large_model_interface.yaml
```

2. Locate the ollama-related configuration

```
#.....
## Offline Large Language Models
# Ollama Configuration
ollama_host: "http://localhost:11434" # Ollama server address
ollama_model: "llava" # Key: Replace this with your downloaded multimodal model,
such as "llava"
#.....
```

Note: Ensure that the model specified in the configuration parameters (such as `llava`) can handle multimodal input.

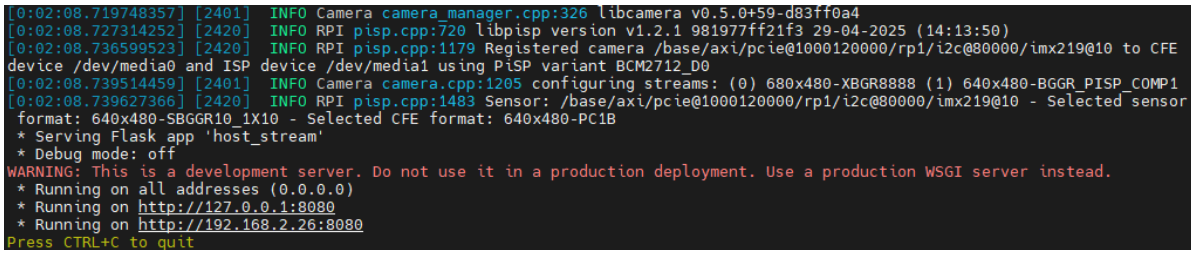
3.1.3 How to Use a CSI Camera

Note: This step is required for Raspberry Pi 5. ORIN and RDK X5 motherboards can skip this step.

First, ensure that `camera_type` in `yahboom.yaml` is set to `csi`.

Then, connect the CSI camera to the motherboard's J4 port. Next, enter the following command in the host machine's terminal:

```
python ~/host_stream.py
```



```
[0:02:08.719748357] [2401]  INFO Camera camera_manager.cpp:326 libcamera v0.5.0+59-d83ff0a4
[0:02:08.727314252] [2420]  INFO RPI pisp.cpp:720 libpisp version v1.2.1 981977ff21f3 29-04-2025 (14:13:50)
[0:02:08.736599523] [2420]  INFO RPI pisp.cpp:1179 Registered camera /base/axi/pcie@1000120000/rp1/i2c@80000/imx219@10 to CFE
device /dev/media0 and ISP device /dev/media1 using PiSP variant BCM2712_D0
[0:02:08.739514459] [2401]  INFO Camera camera.cpp:1205 configuring streams: (0) 680x480-XBGR8888 (1) 640x480-BGGR_PISP_COMP1
[0:02:08.739627366] [2420]  INFO RPI pisp.cpp:1483 Sensor: /base/axi/pcie@1000120000/rp1/i2c@80000/imx219@10 - Selected sensor
 format: 640x480-SBGGR10_1X10 - Selected CFE format: 640x480-PC1B
 * Serving Flask app 'host_stream'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:8080
 * Running on http://192.168.2.26:8080
Press CTRL+C to quit
```

After successful activation, you can continue with the following steps.

3.2 Starting and Testing the Functionality (Text Input Mode)

Note: The performance will be poor when using a model with a small number of parameters or when running at a very low memory usage. For a better experience with this function, please refer to the corresponding section in <Online Large Model (Text Interaction)>.

1. **Start the `largemodel` main program (text mode)**:
   Open a terminal and enter Docker, then run the following command:

```
ros2 launch largemodel largemodel_control.launch.py text_chat_mode:=true
```

2. **Send a text command**:
   Open another new terminal, enter the same Docker container, and run the following

command:

```
ros2 run text_chat text_chat
```

Then start typing: "What do you see?"

3. **Observing the results:**

In the first terminal running the main program, you will see log output showing that the system received the text command, invoked the `seewhat` tool, and finally printed the text description of the desktop generated by the LLaVA model.

---

4. Common Problems and Solutions

Problem 1: Logs display "Failed to call ollama vision model" or connection refused.

Solution:

1. **Check the Ollama service**: Run `ollama list` in the terminal to ensure that `llava` (or your configured model) has been downloaded and the Ollama service is running.
2. **Check the configuration files**: Carefully check the configurations in `yahboom.yaml` and `large_model_interface.yaml` for correctness, especially the values of `llm_platform` and `ollama_model`.

Problem 2: The `seewhat` tool returns "Unable to open camera" or fails to take a picture.

Solution:

1. **Check device connection**: Run `ls /dev/video*` to ensure the system can detect your USB camera. For CSI cameras, ensure they are connected to the J4 port and that `host_stream.py` is running on the host machine.
2. **Permission Issues**: Try testing the camera using applications such as `cheese` or `guvcview`. If sudo works but regular users cannot, it may be a problem with udev rules or user group permissions.