4. Multimodal Visual-Voice Interaction

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 eyes that can think.

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

1.2 Brief Description of Implementation Principles

The basic principle is to input two different types of information—**images (visual information)** and **text (language 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.
2. **Text Encoding:** Simultaneously, the user's question (e.g., "What's on the table?") is also converted into text vectors.
3. **Cross-Modal Fusion:** The most crucial step is the fusion of image and text vectors in a special "attention layer." Here, the model learns to "focus" on the parts of the image 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) generates a descriptive text as the answer based on the fused information.

In short, it **uses text to "highlight" the corresponding parts of the image, and then uses language to describe those "highlighted" parts.**

2. Project Architecture

Key Code

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

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

```python
# 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:
            # ... (Build Prompt)

            # Force use of a plain text system prompt with a clean, one-time context.
            simple_context = [{
                "role": "system",
                "content": "You are an image description assistant. ..."
            }]

            result = self.node.model_client.infer_with_image(image_path, scene_prompt, message=simple_context)
            # ... (Processing Result)
            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 according to 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 for other platforms)
        # ...
        return {'response': response_content, 'messages': self.messages.copy()}
```

Code Analysis

The implementation of this feature 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 Online LLM

Raspberry Pi 5 requires entering a Docker container; RDK X5 and Orin controllers do not require this:

```
./ros2_docker.sh
```

If you need to enter the same Docker container and run other commands later, simply enter ./ros2_docker.sh again in the host terminal.

1. **Next, you need to update the key in the configuration file. Open the model interface configuration file `large_model_interface.yaml`**:

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

2. **Enter your API Key**:
   Find the corresponding section and paste the API Key you just copied. This example uses the Tongyi Qianwen configuration.

   ```
   # large_model_interface.yaml

   ## Thousand Questions on Tongyi
   qianwen_api_key: "sk-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx" # Paste your Key
   qianwen_model: "qwen-vl-max-latest" # You can choose the model as needed,
   such as qwen-turbo, qwen-plus
   ```

3. **Select the online platform you want to use**:

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

4. **选择要使用的在线平台**:
   Change the `llm_platform` parameter to the platform name you want to use.

   ```
   # yahboom.yaml

   model_service:
     ros__parameters:
       # ...
       llm_platform: 'tongyi'  #Optional platforms: 'ollama', 'tongyi',
   'spark', 'qianfan', 'openrouter'
   ```

After modifying the configuration file, you need to recompile and source it in the workspace:

```
cd ~/yahboom_ws
colcon build && source install/setup.bash
```

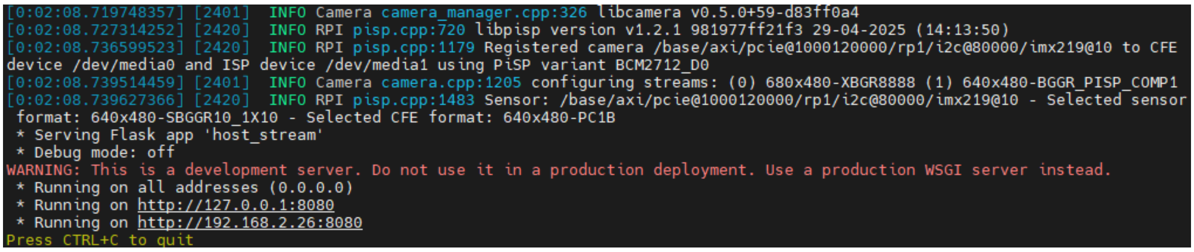
How to Use a CSI Camera

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

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

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

1. **After entering the Docker container, start the `largemodel` main program:**

   Open a terminal and run the following command:

```
ros2 launch largemodel largemodel_control.launch.py
```

2. **Testing:**

- **Wake-up:** Speak into the microphone: "Hello, Xiaoya."
- **Dialogue:** After the speaker responds, you can say: `What do you see?`
- **Observe the Logs:** In the terminal running the `launch` file, you should see:

1. The ASR node recognizes your question and prints it out.
2. The `model_service` node receives the text, calls LLM, and prints LLM's response.

- **Hear the Response:** Later, you should hear the response from the speaker.