4. Multimodal Visual Understanding Voice Interaction

Note: Raspberry Pi 5 2GB/4GB, RDK X5 4GB, and Jetson Orin Nano 4GB versions cannot run offline due to performance limitations. 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 the robot to not only "see" a pixel matrix, but also to truly "understand" the content, objects, scenes, and relationships between them in an image. This is like giving the robot a pair of thinking eyes.

The core tool for this function is `seewhat`. When the user gives a command such as "Look at what's here," the system calls this tool, triggering a series of background operations, and ultimately feeding back the AI's analysis results 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 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 the fusion of image and text vectors in a special "attention layer." Here, the model learns to "pay attention" to the parts of the image

relevant to the question. For example, when asked about "table," the model will focus more on 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 the 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)

            # 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 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 according to the configuration.

```
# 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

This feature's implementation 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 business of the visual understanding function. It encapsulates the entire "seeing" action process: first, it calls the `capture_frame` method to obtain an image, then calls `_get_actual_scene_description` to prepare a prompt for the model to analyze the image.
    - The most critical step is calling the `infer_with_image` method of the model interface layer. It does not care about the underlying model; it only passes the two core data elements, "image" and "analysis instructions," to the model interface layer.
    - Finally, it packages the analysis results (plain text descriptions) received from the model interface layer into a structured dictionary and returns them. 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 "dispatching center." Its primary responsibility is to check the current platform configuration (`self.llm_platform`) and, based on the configuration, dispatch tasks to specific handlers (such as `ollama_infer` or `tongyi_infer`).
   - This layer is key to adapting to different AI platforms. All platform-specific operations (such as data encoding and API call formats) are encapsulated within their respective handlers.
   - In this way, the business logic code in `tools_manager.py` remains unchanged to support a variety of different large-model backend services. It simply interacts with the unified, stable `infer_with_image` interface.

In summary, the `seewhat` tool's execution flow demonstrates a clear separation of responsibilities: `ToolsManager` defines the "what" (acquiring an image and requesting analysis), while `model_interface` defines the "how" (selecting the appropriate model platform based on the current configuration and interacting with it). This makes the tutorial's parsing universal; the core code logic remains consistent regardless of whether the user is online or offline.

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.

Raspberry Pi 5 requires entering a Docker container, while RDK X5 and Orin controllers do not:

```
./ros2_docker.sh
```

If you need to enter other commands in the same Docker container later, simply enter ./ros2_docker.sh again in the host 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 it's 'ollama'
       regional_setting : "China"
   ```

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

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

1.Open the file in a terminal

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

2.Find the Ollama-related configuration

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

Note: Please ensure that the model specified in the configuration parameters (e.g., `llava`) can handle multimodal input.

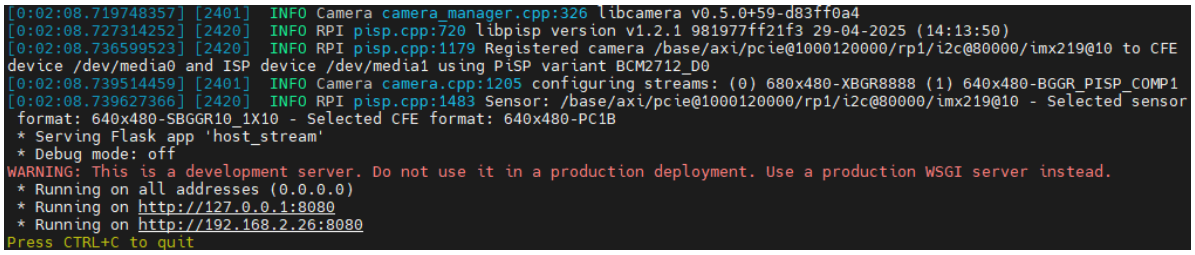
3.1.3 How to Use a CSI Camera

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

Note: The performance will be worse when using a model with a small number of parameters or when memory usage is at its limit. For a better experience with this feature, please refer to the corresponding chapter in <Online Large Model (Text Interaction)>

1. **Start the `largemodel` main program**:

   Run the following command:

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

2. After successful initialization, say the wake word and then ask: What do you see? Or describe the current environment.

3. **Observe the results**:

   In the first terminal running the main program, you will see log output showing that the system received the text command, called the `seewhat` tool, and finally printed the text description generated by the LLaVA model. The speaker will also announce the generated result.

---

4. Common Problems and Solutions

4.1 Very Slow Response

Problem: After asking a question, it takes a long time to get a voice response.

Solution: The inference cost of multimodal models is much higher than that of pure text models, so higher latency is normal.

1. **Use a smaller model:** Try using a lighter multimodal model in `large_model_interface.yaml`.