4. Multimodal Visual Understanding and 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. Code Analysis

Key Code

1. Tool 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:
            # ... (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.

```
# 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. """
        # ... (Preparing the message)
        try:
            # The value of self.llm_platform determines which specific
implementation to call.
            if self.llm_platform == 'ollama':
                response_content = self.ollama_infer(self.messages,
image_path=image_path)
            elif self.llm_platform == 'tongyi':
                # ... Logic of calling the generalized model
                pass
            # ... (Logic for 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, maintaining consistent core code logic 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.

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: 'en'                      # Large model interface language
    useolinetts: True                   # This option is invalid in text mode
and can be ignored

    # Large model configuration
    llm_platform: 'ollama'               # Critical: Ensure this is 'ollama'
```

3.1.2 Configure the model interface (`large_model_interface.yaml`)

This file defines which visual model is used 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 Recompile

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

3.2 Startup and Testing

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

Open a terminal and 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. **Observations:**

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 then 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:** In `large_model_interface.yaml`, try using a lighter version of the `llava` model.