Multimodal visual understand speech interaction

Multimodal visual understand speech interaction

- 1. Concept Introduction
  - 1.1 What is "Visual Understanding"?
  - 1.2 Implementation Principle Overview
- 2. Project Architecture
  - Key Code
    - 1. Tool Layer Entry (`largemodel/utils/tools_manager.py`)
    - 2. Model Interface Layer (`largemodel/utils/large_model_interface.py`)
  - Code Analysis
- 3.Practical Operation
  - 3.1 Configuring Online LLM
  - 3.2 Start and test the functionality

1. Concept Introduction

1.1 What is "Visual Understanding"?

In the `largemodel` project, the **multimodal visual understanding** feature enables robots to go beyond simply "seeing" a matrix of pixels and truly "understand" the content, objects, scenes, and relationships within an image. This is like giving robots a pair of thinking eyes.

The core tool for this feature is **seewhat**. When a user issues a command like "see what's here," the system invokes this tool, triggering a series of background operations that ultimately provide the user with AI-generated analysis of the live image in natural language.

1.2 Implementation Principle Overview

The basic principle is to input two different types of information—**image (visual information)** and **text (linguistic information)**—into a powerful multimodal large model (such as LLaVA).

1. **Image Encoding**: The model first uses a vision encoder to convert the input image into computer-interpretable digital vectors. These vectors capture image features such as color, shape, and texture. 2. **Text Encoding**: Simultaneously, the user's question (e.g., "What's on the table?") is converted into a text vector.
2. **Cross-modal Fusion**: In the most crucial step, the model fuses the image vector and the text vector 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 areas of the image that match the characteristics of a table.
3. **Answer Generation**: Finally, a large language model (LLM) generates a descriptive text answer based on this fused information.

Simply put, this involves **highlighting the corresponding parts of the image with text, and then describing the highlighted parts with language**.

2. Project Architecture

Key Code

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

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

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

class ToolsManager:
    # ...

    def seewhat(self):
        """
        Capture camera frame and analyze environment with AI model.
        捕获摄像头画面并使用AI模型分析环境。

        :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. / 使用隔离的上下文进行图像分
析。
            analysis_text = self._get_actual_scene_description(image_path)

            # Return structured data for the tool chain. / 为工具链返回结构化数据。
            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.
        获取捕获图像的AI生成场景描述。

        :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. / 强制使用纯文本系统提示和干净的一次性上下文。
            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 based on 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. / 统一的图像推理接口。"""
        # ... (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 parsing universal, and its core code logic is consistent regardless of whether the user is in online or offline mode.

3.Practical Operation

3.1 Configuring Online LLM

1. **First, get the API Key from any platform in the previous tutorial**
2. **Then you need to update the key in the configuration file and open the model interface configuration file** `large_model_interface.yaml`:

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

3. **Fill in your API Key**:
   Find the corresponding part and paste the API Key you just copied into it. Here we take Tongyi Qianwen configuration as an example

   ```
   # 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
   ```

4. **Open the main configuration file** `yahboom.yaml`:

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

5. **Select the online platform you want to use**:
   Change the `llm_platform` parameter to the platform name you want to use.

6.
   ```
   # yahboom.yaml

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

3.2 Start and test the functionality

1. **start up**:

Note: The startup commands for CSi cameras and USB microphone cameras are different. Please run the appropriate command for your camera.

CSI Camera

Start udp video streaming (host)

```
./start_csi.sh
```

Enter the CSI camera docker (host machine)

```
./run_csi_docker.sh
```

Start topic conversion (docker)

```
python3 ~/temp/udp_camera_publisher.py
```

View container id

```
docker ps
```

According to the container ID shown above, multiple terminals enter the same docker

```
docker exec -it  container_id  /bin/bash
```

Run the following command to enable voice interaction:

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

USB Camera

Enter the USB camera docker (host machine)

```
./run_usb_docker.sh
```

Run the following command to enable voice interaction:

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

3. **Test**:
   - **Wake up**: Say "Hi, yahboom" into the microphone.
   - **Talk**: After the speaker responds, you can say, "What do you see?"
   - **Observe the log**: In the terminal running the `launch` file, you should see the following:
   1. The ASR node recognizes your question and prints it.

2. The `model_service` node receives the text, calls the LLM, and prints the LLM's response.

- **Listen for the answer**: After a while, you should hear the answer from the speaker.