3. Multimodal Visual Localization Application

1. Concept Introduction

1.1 What is "Multimodal Visual Localization"?

Multimodal visual localization is a technology that combines multiple sensor inputs (such as cameras, depth sensors, IMUs, etc.) and algorithm processing techniques to achieve precise identification and tracking of device or user position and orientation in the environment. This technology not only relies on a single type of sensor data, but integrates information from different perception modes, thereby improving the accuracy and robustness of localization.

1.2 Brief Implementation Principle

1. **Cross-modal Representation Learning**: To enable LLMs to process visual information, a mechanism needs to be developed to convert visual signals into a form that the model can understand. This may involve using convolutional neural networks (CNNs) or other architectures suitable for image processing to extract features and map them to the same embedding space as text.
2. **Joint Training**: By designing appropriate loss functions, text and visual data can be trained simultaneously within the same framework, enabling the model to learn to establish connections between these two modalities. For example, in a question-answering system, it can provide answers based on both provided text questions and relevant image content.
3. **Vision-guided Language Generation/Understanding**: Once effective cross-modal representation is established, visual information can be used to enhance the functionality of language models. For example, when given a photo, the model can not only describe what's happening in the image, but also answer specific questions about the scene, and even execute instructions based on visual cues (such as navigating to a location).

2. Code Analysis

Key Code

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

The `visual_positioning` function in this file defines the execution flow of this tool, specifically how it constructs a Prompt containing target object names and format requirements.

```
# From largemodel/utils/tools_manager.py
class ToolsManager:
    # ...
    def visual_positioning(self, args):
        """
        Locate object coordinates in image and save results to MD file.

        :param args: Arguments containing image path and object name.
        :return: Dictionary with file path and coordinate data.
        """
        self.node.get_logger().info(f"Executing visual_positioning() tool with args: {args}")
        try:
            image_path = args.get("image_path")
            object_name = args.get("object_name")
            # ... (Path fallback mechanism and parameter checking)

            # Construct a prompt asking the large model to identify the coordinates of the specified object.
            if self.node.language == 'zh':
                prompt = f"Please carefully analyze this image and find the position of all {object_name}..."
            else:
                prompt = f"Please carefully analyze this image and find the position of all {object_name}..."

            # ... (Build independent message context)

            result = self.node.model_client.infer_with_image(image_path, prompt, message=message_to_use)

            # ... (Process and parse returned coordinate text)

            return {
                "file_path": md_file_path,
                "coordinates_content": coordinates_content,
                "explanation_content": explanation_content
            }
        # ... (Error handling)
```

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-related tasks.

```
# 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:
            # Decide 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 Tongyi model
                pass
            # ... (Logic for other platforms)
        # ...
        return {'response': response_content, 'messages': self.messages.copy()}
```

Code Analysis

The core of the visual localization function lies in **guiding the large model to output structured data through precise instructions**. It also follows the layered design of the tool layer and model interface layer.

1. **Tool Layer (`tools_manager.py`)**:
    - The `visual_positioning` function is the business core of this feature. It receives two key parameters: `image_path` (image path) and `object_name` (name of the object to locate).
    - The most core operation of this function is **building a highly customized Prompt**. It not only simply requests the model to describe the image, but embeds `object_name` into a carefully designed template, explicitly requiring the model to "locate the position of every {object_name} in the image" and implicitly or explicitly requiring results to be returned in a specific format (such as coordinate arrays).
    - After building the Prompt, it calls the `infer_with_image` method of the model interface layer, passing both the image and this customized instruction.
    - After receiving the returned text from the model interface layer, it also needs to perform **post-processing**: use methods like regular expressions to parse precise coordinate data from the model's natural language reply.
    - Finally, it returns the parsed structured coordinate data to the upper-level application.
2. **Model Interface Layer (`large_model_interface.py`)**:
    - The `infer_with_image` function still plays the role of "dispatch center". It receives the image and Prompt from `visual_positioning`, and distributes tasks to the correct backend model implementation based on the current configuration (`self.llm_platform`).
    - For visual localization tasks, the responsibilities of the model interface layer are basically the same as for visual understanding tasks: correctly package image data and text instructions, send them to the selected model platform, and then return the text results unchanged to the tool layer. All platform-specific implementation details are encapsulated in this layer.

In summary, the general flow of visual localization is: `ToolsManager` receives the target object name and builds a precise Prompt requesting coordinate return -> `ToolsManager` calls the model interface -> `model_interface` packages the image and this Prompt together, and sends it to the corresponding model platform based on configuration -> Model returns text containing coordinate information -> `model_interface` returns the text to `ToolsManager` -> `ToolsManager` parses the text, extracts structured coordinate data and returns it. This process demonstrates how to use Prompt Engineering techniques to make general visual large models complete more specific and structured tasks.

3. Practical Operations

3.1 Configure Online LLM

1. **First obtain API Key from OpenRouter platform**
2. **Then 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
   ```

3. **Enter your API Key**:
   Find the corresponding section and paste the API Key you just copied. Here we use Tongyi Qianwen configuration as an example

   ```
   # large_model_interface.yaml

   # OpenRouter platform configuration
   openrouter_api_key: "sk-xxxxxxxxxxxxxxxxxxxxxxxxxxxx"
   openrouter_model: "nvidia/nemotron-nano-12b-v2-vl:free" # Model to use, e.g., "google/gemini-pro-vision"
   ```

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

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

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

   ```
   # yahboom.yaml

   model_service:
     ros__parameters:
       # ...
       llm_platform: 'openrouter'              # Currently selected large model platform
       # Available platforms: 'ollama','openrouter'
   ```

Recompile

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

3.2 Start and Test Function

1. **Prepare image file**:

   Place a test image file in the following path: `/home/sunrise/yahboom_ws/src/largemodel/resources_file/visual_positioning`

   Then rename the image to `test_image.jpg`

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

   Open a terminal, then run the following command:

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

3. **Send text commands**:
   Open another terminal and run the following command,

   ```
   ros2 run text_chat text_chat
   ```

   Then start typing text: "Analyze the position of the dinosaur in the image".

4. **Observe results**:
   In the first terminal running the main program, you will see log output showing that the system received the command, called the `visual_positioning` tool, prompted that visual_positioning execution is complete, and saved the coordinates to the document.

   We can find this document in the ~/yahboom_ws/src/largemodel/resources_file/visual_positioning path.