8. Multimodal Visual Localization Applications

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 algorithmic processing techniques to achieve precise identification and tracking of the location and posture of devices or users in their environment. This technology does not rely solely on a single type of sensor data but integrates information from different perceptual modalities, thereby improving the accuracy and robustness of localization.

1.2 Brief Description of Implementation Principles

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 image-processing-appropriate architectures to extract features and map them into the same embedding space as the text.
2. **Joint Training**: By designing an appropriate loss function, text and visual data can be trained simultaneously within the same framework, allowing the model to learn to establish connections between the two modalities. For example, in a question-answering system, answers can be provided based on both text questions and related image content.
3. **Visually Guided Language Generation/Understanding**: Once an effective cross-modal representation is established, visual information can be used to enhance the functionality of the language model. For instance, given a photograph, the model can not only describe what is 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 Point (`largemodel/utils/tools_manager.py`)

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

```python
# 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"请仔细分析这张图片，用一个个框定位图像每一个{object_name}的位置..."
            else:
                prompt = f"Please carefully analyze this image and find the position of all {object_name}..."

            # ... (Build an independent message context)

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

            # ... (Process and parse the 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.

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

1. **Tool Layer (`tools_manager.py`):**

- The `visual_positioning` function is the core of this function's business logic. It receives two key parameters: `image_path` (image path) and `object_name` (the name of the object to be positioned).
- The most crucial operation of this function is **constructing a highly customized Prompt**. It doesn't simply request the model to describe the image; instead, it embeds `object_name` into a carefully designed template, explicitly instructing the model to "position each {object_name} in the image," and implicitly or explicitly requesting the return of results in a specific format (such as a coordinate array).
- After constructing the Prompt, it calls the `infer_with_image` method of the model interface layer, passing the image and this customized instruction along with it.
- After receiving the returned text from the model interface layer, it still needs to perform **post-processing**: using methods such as regular expressions to parse precise coordinate data from the model's natural language response.
- Finally, it returns the parsed structured coordinate data to the upper-layer application.

2. **Model Interface Layer (`large_model_interface.py`):**

- The `infer_with_image` function still plays the role of the "dispatch center." It receives the image and prompt from `visual_positioning` and distributes the task to the correct backend model implementation based on the current configuration (`self.llm_platform`).
- For visual positioning tasks, the model interface layer's responsibilities are essentially the same as for visual understanding tasks: correctly package the image data and text instructions, send them to the selected model platform, and then return the text results intact to the tool layer. All platform-specific implementation details are encapsulated in this layer.

In summary, the general process of visual localization is as follows: `ToolsManager` receives the name of the target object and constructs a precise Prompt that requests the return of coordinates -> `ToolsManager` calls the model interface -> `model_interface` packages the image and the Prompt together and sends it to the corresponding model platform according to the configuration -> The model returns text containing coordinate information -> `model_interface` returns the text to `ToolsManager` -> `ToolsManager` parses the text, extracts the structured coordinate data, and returns it. This process demonstrates how Prompt Engineering technology allows general-purpose large-scale visual models to perform more specific and structured tasks.

3. Practical Operation

3.1 Configuring Online LLM

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

```
./ros2_docker.sh
```

If you need to enter other commands within the same Docker container later, simply type `./ros2_docker.sh` again in the host machine's terminal.

1. **Then 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. Here, we'll use the Tongyi Qianwen configuration as an example.

```
# large_model_interface.yaml

## Thousand Questions on Tongyi
qianwen_api_key: "sk-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx" # 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 to use:**

Modify the `llm_platform` parameter to the name of the platform 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
```

3.2 Starting and Testing the Feature

1. **Preparing the Image File**:

Place the image file to be tested in the following path: `~/yahboom_ws/src/largemodel/resources_file/visual_positioning`

Then name the image `test_image.jpg`

2. Starting the `largemodel` main program:

Open a terminal, enter the Docker container, and run the following command:

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

3. **Sending Text Commands**:

Open another terminal, enter the same Docker container, and run the following command:

```
ros2 run text_chat text_chat
```

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

4. **Observations:**

In the first terminal running the main program, you will see log output showing that the system received the command, invoked the `visual_positioning` tool, and indicated that `visual_positioning` had completed execution and saved the coordinates to a file.

This file can be found at ~/yahboom_ws/src/largemodel/resources_file/visual_positioning.