main.py

A **camera object** is created and configured to capture video at a resolution of 640x480. The camera is connected at index `4`, which you can adjust based on your system.

Main Loop

The `while` loop keeps the system running, continuously capturing frames from the camera and processing them.

Frame Capture

The camera captures a frame (`frame`), and if unsuccessful, it exits the loop.

Object Detection

Objects in the frame are detected (e.g., pedestrians, vehicles) using a function `detect_objects`. This might utilize a pre-trained object detection model such as YOLO or SSD. While the results are not explicitly visualized in this code, the function might flag obstacles.

Lane Detection and Processing

Lane lines are detected using the `lane_detection_process`. This function likely involves:

- Edge detection (e.g., Canny Edge Detection).
- Hough Line Transform to find lane-like lines.
- Filtering lines based on orientation and position.

Detected lane lines are drawn on the `frame` for visualization using the `draw_lane_lines` function.

Forward Line Angle

This calculates the angle of the forward lane line relative to the vehicle's center, indicating whether the lane curves left, right, or is straight.

Midpoint and Turn Direction

The **midpoint** of the lane is calculated to determine the vehicle's lateral position within the lane.
The **turn direction** is determined by comparing the lane midpoint with the image's center:

- **Left** if midpoint is left of center.

- **Right** if midpoint is right of center.
- **Straight** if midpoint aligns with the center.
- If no lanes are detected, no turn is specified.

Motor Control

The robot's motors are controlled based on the turn direction:

- **Turn left**: Adjust motor speeds to steer left.
- **Turn right**: Adjust motor speeds to steer right.
- **Move forward**: Maintain forward motion.
- **Stop motors**: Halt the robot if no lanes are detected.

Visualization

The lane midpoint is marked with a blue circle on the frame.
The turn direction is displayed as text.

Display Frame

The processed frame is displayed in a window.
The program exits when the `q` key is pressed.

Cleanup

Stops the motors, releases the camera resource, and closes any OpenCV display windows.

motor_control.py

def set_motor_speeds

`set_motor_speeds`: Central function that controls the individual motor speeds.
Direction Control:

- Positive values move motors forward.
- Negative values move motors backward.

Speed Control:

- Adjusting the speed values changes the movement speed or turning behavior.

Integration:

- Replace the `print` statements with actual commands to interface with hardware.

utils.py

def average_lanes

Input: `lane_history` (a list of tuples with lane positions, where each tuple contains two lane lines: left and right).

Process:

- It separates the left and right lane lines that are not `None`.
- It calculates the mean (average) of the left and right lane lines based on all the valid lines.

Output:

- Returns two values: the average of the left lane and the average of the right lane. If no valid lane is found, it returns `None` for the respective lane.

def average_slope_intercept

Input: `lines` (a list of detected lines, where each line is represented by four coordinates: `(x1, y1, x2, y2)`).
Process:

- The function iterates through all lines and computes their slope and intercept using the formula:
    - `slope = (y2 - y1) / (x2 - x1)`
    - `intercept = y1 - (slope * x1)`
    - It calculates the length of each line for weighting purposes.
- Lane lines are separated into left and right based on the slope. A negative slope indicates the left lane, and a positive slope indicates the right lane.
- Weighted averages for both slope and intercept are calculated for the left and right lane lines.

Output: Returns a tuple `(left_lane, right_lane)` with the average `(slope, intercept)` of both lanes. If there are no valid lines, it returns `None`.

def pixel_points

Input: `y1` and `y2` (the y-coordinates where you want to calculate the x-coordinates of the lane line), `line` (a tuple of slope and intercept of the lane).
Process:

- It calculates the x-coordinate at the given `y` values by rearranging the equation of a line: `x = (y - intercept) / slope`.
- If the slope is very small or zero, the function returns `None` because this means the line is nearly horizontal.

Output: It returns the pixel coordinates `(x1, y1)` and `(x2, y2)` as tuples representing the start and end points of the lane line. If the slope is nearly zero or invalid, it returns `None`.

def determine_turn

Input: `midpoint` (the x-coordinate of the midpoint between the two lane lines), `image_center_x` (the x-coordinate of the image center).
Process:

- It calculates the deviation of the midpoint from the center of the image. If the deviation exceeds a threshold, it determines if the vehicle is turning left or right.
- If the midpoint is close to the center, it returns "straight."

Output: Returns a string: `"left"`, `"right"`, or `"straight"`, based on the midpoint deviation.

def forward_region_of_interest

Input: `image` (the original input image).
Process:

- It calculates a region from the bottom of the image (`height // 4` to `height`), with a width that is `width // 4`.
- This helps focus on the relevant portion of the image where lane lines typically appear.

Output: Returns the cropped image corresponding to the region of interest.

def calculate_forward_line_angle

Input: `image` (the original image).
Process:

- It extracts the region of interest from the bottom part of the image using `forward_region_of_interest()`.
- Converts the region to grayscale and detects edges using the Canny edge detector.

- Then, it uses Hough Transform to detect lane lines and calculates the angle of each line using the `atan2()` function to get the angle in radians, which is converted to degrees.

Output: Returns the calculated angle of the lane line in degrees. If no lines are detected, it returns `None`.

def get_lane_midpoint

Input: `left_line` and `right_line` (each containing the coordinates of the bottom of the lane), `image_width` (width of the image).
Process:

- If both lanes are detected, the midpoint is calculated as the average of the x-coordinates of the left and right lane bottoms.
- If only one lane is detected, an approximation is made by adjusting the midpoint based on the side where the lane is detected.
- It ensures that the midpoint stays within the bounds of the image.

Output: Returns the calculated midpoint. If no valid lanes are detected, it returns `None`.

def draw_lane_lines

Input: `image` (the original image), `lines` (a list of lane lines, each containing a tuple of start and end points), `color` (the color of the lane lines), `thickness` (thickness of the lane lines).
Process:

- It creates a blank image (`line_image`) and draws the lane lines on this blank image using `cv2.line()`.
- Then it overlays the drawn lines onto the original image using `cv2.addWeighted()`, blending the lines with the original image.

Output: Returns the original image with the lane lines drawn on it.

def calculate_distance

Input: `H_real` (the real height of the object), `H_image` (the height of the object in the image), `focal_length` (the focal length of the camera).
Process:

- Uses the formula: `distance = (H_real * focal_length) / H_image` to calculate the distance between the object and the camera.

Output: Returns the calculated distance. If the image height is zero or invalid, it returns `None`.

lane_detection.py

def lane_detection_process

Process:

1. **Convert to HSV and Detect White:**
    - The frame is converted from BGR to HSV color space using `cv2.cvtColor()`. HSV is better for color-based segmentation.
    - A white color mask is created to isolate white lane lines by defining a range for white in HSV (`lower_white` and `upper_white`).
    - `cv2.inRange()` is used to generate a binary mask where white regions in the HSV frame are highlighted.
    - `cv2.bitwise_and()` applies the mask to the original frame to keep only the white regions.
2. **Convert to Grayscale and Detect Edges:**
    - The resulting masked frame is converted to grayscale for edge detection using `cv2.cvtColor()`.
    - Edges are detected using the Canny edge detector (`cv2.Canny()`), which highlights the boundaries of white regions.
3. **Region of Interest (ROI):**
    - The lower half of the frame is defined as the region of interest to focus on lane lines.
    - A polygon mask is created using `cv2.fillPoly()` to isolate the lower portion of the frame.
4. **Apply Mask to Edges:**
    - The polygon mask is applied to the edge-detected frame using `cv2.bitwise_and()` to retain only edges within the ROI.
5. **Detect Lines Using Hough Transform:**
    - The Hough Line Transform (`cv2.HoughLinesP()`) is applied to detect line segments in the masked edge frame.
6. **Extract Lane Lines:**
    - The detected lines are passed to the `lane_lines()` function to calculate and return the left and right lane coordinates.

Output:

- Returns the coordinates of the detected left and right lane lines.

def lane_lines

Process:

1. **Check for Valid Lines:**
    - If no lines are detected (`lines` is `None` or empty), the function returns `None` for both lanes.
2. **Calculate Average Slope and Intercept:**
    - The function calls `average_slope_intercept(lines)` (from `utils`) to separate left and right lanes based on slope and calculate their average slopes and intercepts.
3. **Define Line Endpoints:**
    - The bottom of the image (`y1`) and a point 45% from the bottom (`y2`) are used as vertical bounds for the lanes.
    - Using the calculated slope and intercept, the lane lines are converted into pixel coordinates for these vertical bounds via `pixel_points()`.

Output:

- Returns two tuples, `left_line` and `right_line`, each containing start and end pixel coordinates of the corresponding lane. If a lane is not detected, its value is `None`.

def draw_lane_lines

Process:

1. **Create a Blank Image:**
    - A blank image (`line_image`) is created with the same dimensions as the input `image`.
2. **Draw Lines:**
    - The function iterates through the `lines` (a list of lane line coordinates).
    - If a line is valid (not `None`), it retrieves its start (`pt1`) and end (`pt2`) points, ensuring they are integers.
    - The line is drawn on the blank image using `cv2.line()` with the specified `color` and `thickness`.
3. **Overlay Lines on the Original Image:**
    - The blank image with the drawn lane lines is blended with the original image using `cv2.addWeighted()` to create a final image.

Output:

- Returns the original image with lane lines overlaid.

object_detection.py

def calculate_ground_distance

Parameters:

- `v`: The vertical pixel coordinate of the object's base in the image (e.g., bottom of a bounding box).
- `image_height`: The height of the image in pixels.
- `focal_length_px`: The focal length of the camera in pixels.
- `camera_height`: The height of the camera above the ground in meters.

Process:

1. Compute the vertical offset of the object base from the center of the image (`pixel_offset`).
2. Calculate the angle of depression (`theta`) using the arctangent of the offset divided by the focal length.
3. Use trigonometric relationships to calculate the ground distance:
    - distance=camera_heighttan(θ)\text{distance} = \frac{\text{camera_height}}{\tan(\theta)}distance=tan(θ)camera_height
4. Return the distance if valid (i.e., θ>0\theta > 0θ>0).

Output:

- Returns the ground distance in meters. If the angle is invalid, returns `None`.

def detect_objects

Process:

1. **Perform Inference with YOLO:**
    - The YOLO model (`model`) processes the frame and detects objects.
2. **Iterate Through Detections:**
    - For each detected object (`detection`):
        - Extract the **class ID** (object type) and **confidence score**.
        - Ensure the confidence score meets the threshold (e.g., 0.2).
        - Get the bounding box coordinates (`x1, y1, x2, y2`).
3. **Calculate Ground Distance:**
    - Use the bottom y2y2y2 of the bounding box as the reference point for ground distance calculation.
    - Call `calculate_ground_distance()` to compute the distance to the object based on the camera's height.
4. **Annotate the Frame:**
    - Draw a bounding box around the detected object.

    - Overlay the object's name, confidence score, and calculated distance on the frame.
5. **Take Motor Actions Based on Distance:**
    - **If the distance is between 0.8 and 1.0 meters** and the object is a **person**:
        - Overlay "STOP!" on the frame and call `stop_motors()`.
    - **If the distance is less than 0.8 meters** or the object is a **traffic cone**:
        - Overlay "REVERSE!" on the frame and call `reverse()`.
    - **For other objects within 0.8 to 1.0 meters**:
        - Overlay "TURN!" on the frame (actions can be expanded with additional logic).

Output:

- Returns the annotated video frame, with:
    - Bounding boxes around detected objects.
    - Distance measurements and labels (e.g., "STOP!", "REVERSE!", "TURN!").
    - Visual cues for objects of interest (e.g., people or cones).
- Motor control actions (stopping, reversing, or turning) are triggered as needed.

camera_utils.py

def initialize_camera

Process:

1. **Initialize the Camera:**
    - `cv2.VideoCapture(camera_index)`:
        - Opens a connection to the camera specified by `camera_index`.
        - Returns a `VideoCapture` object for interacting with the camera.
2. **Set Camera Properties:**
    - **Resolution:**
        - `cv2.CAP_PROP_FRAME_WIDTH`: Sets the width of the camera frame to `640` pixels.
        - `cv2.CAP_PROP_FRAME_HEIGHT`: Sets the height of the camera frame to `480` pixels.
    - These settings ensure the captured frames have a fixed resolution.
3. **Check if the Camera is Opened:**
    - `camera.isOpened()`:
        - Returns `True` if the camera is successfully opened.
        - If it fails, the function raises a `ValueError` with the message: `"Error: Could not open camera."`
4. **Return the Camera Object:**
    - If the camera is successfully initialized, the `VideoCapture` object is returned for further use (e.g., capturing frames).

Output:

- Returns a configured `cv2.VideoCapture` object for the specified camera index.