

Road Lane-Line Detection using Python and OpenCV

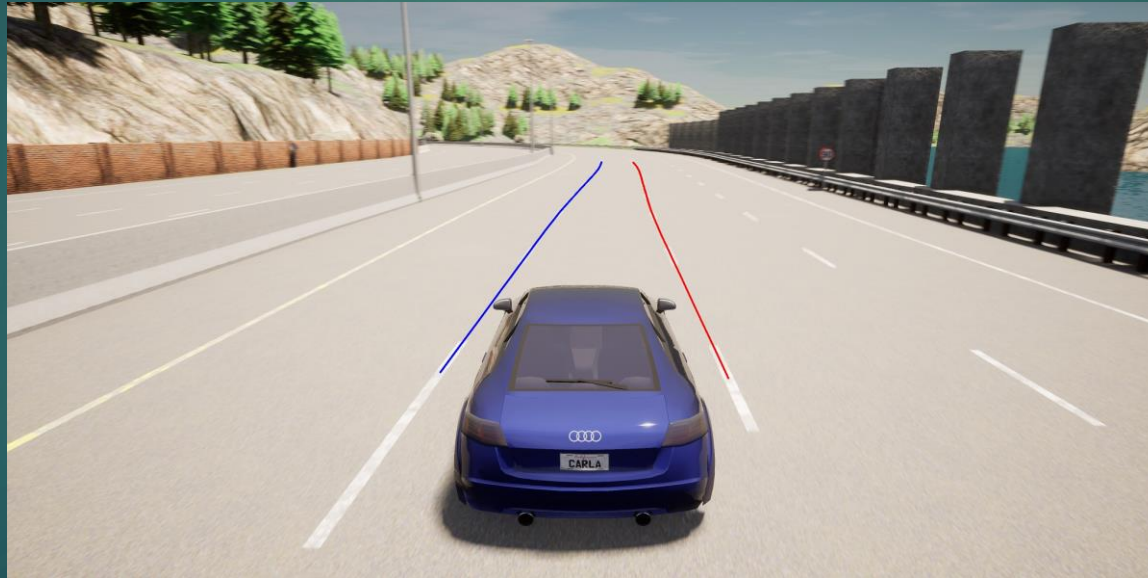
NAME: PRIYANSHU ROHILLA

ENROLLMENT NO. 04616403220

BATCH: B.TECH. CSE 7TH SEM

Problem Statement

AIM: The goal of this project is to develop a robust lane line detection system using computer vision techniques and image processing algorithms. The system should be capable of accurately detecting and highlighting lane lines in real-time video or images captured by a vehicle's camera.



Implementation

We have a function called '*detect_lanes()*' which will detect lanes in a given frame. What happens inside this function?

1. GRAYSCALING

First the frame is converted into grayscale so as to process it easily and faster because it deals with only two colors instead of three (RGB). Converts the input frame to grayscale, simplifying the image for edge detection.

Code: **gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)**

2. GAUSSIAN BLUR

Then Gaussian Blur is applied in the frame so as to reduce the noise and smoothen it.

Code: **blurred = cv2.GaussianBlur(gray, (5, 5), 0)**

3. CANNY EDGE DETECTION

Canny edge detection is applied using OpenCV to detect the edges of the lanes, and a masking function was applied to eliminate distractions such as trees, rocks, and power lines. It uses the canny algorithm.

Code: **edges = cv2.Canny(blurred, 150, 250)**

4. REGION OF INTEREST (ROI) MASKING

Creates a mask to define the region of interest (the area where lane lines are expected) by using a polygon defined by 'region_of_interest_vertices'. This isolates the area for lane detection.

5. APPLYING MASK TO EDGES

Applies the mask to the detected edges, focusing only on the region of interest.

Code: **masked_edges = cv2.bitwise_and(edges, mask)**

6. Hough Transform

Utilizes the Hough Transform to detect lines in the masked edge image, tuning parameters like resolution ('rho'), angle ('theta') and thresholds ('threshold', 'minLineLength', 'maxLineGap').

Code: **lines = cv2.HoughLinesP(masked_edges,rho=6,theta=np.pi / 60,threshold=230,lines=np.array([]),minLineLength=40,maxLineGap=25,)**

Then we have a function called '`draw_lines()`' which will draw detected lane lines on an image.

1. Checking for Valid Lines: `if lines is not None::` Checks if there are detected lines in the input. If there are no lines detected (`None`), the function won't attempt to draw anything.
2. Looping Through Detected Lines: `for line in lines::` Iterates through each detected line in the lines array.
3. Extracting Line Endpoints: `x1, y1, x2, y2 = line[0]`: Extracts the coordinates of the endpoints of the detected line.
4. Drawing Lines on the Image: `cv2.line(image, (x1, y1), (x2, y2), (0, 255, 0), 5)`: Draws a line on the input image using the endpoints `(x1, y1)` and `(x2, y2)`. `(0, 255, 0)` represents the color of the line in BGR format (here, it's green). `5` specifies the thickness of the line in pixels.
5. Result: The function draws each detected line onto the provided image using the specified color and thickness.

RESULT



Conclusion and Future Works

The lane detection script employs fundamental OpenCV techniques for basic lane identification. Future improvements could focus on advanced lane tracking algorithms for stability, integrating machine learning models for robustness, and dynamic region-of-interest adaptation. Enhancements to handle diverse road conditions, optimize real-time performance, and create a user-friendly interface would amplify its practicality. Further testing on varied datasets and deployment in ADAS or autonomous vehicles could validate its efficacy. Integrating color-based detection, perspective transformations, and curvature analysis would refine accuracy. Overall, refining algorithms, expanding datasets, and optimizing for real-time performance would enhance its adaptability and reliability in real-world driving scenarios.