EE 475 Final Project
Vision-Based Autonomous Navigation Assistance
Bogazici Campus Route

Enes Kuzuoğlu 2021401210
Mehmet Emin Algül 2021401258
Word count: 1312

December 2025

Introduction

This project develops a computer vision-based navigation assistance system for an autonomous shuttle operating between Boğaziçi University's North and South Campuses. The work focuses on real-world challenges such as varying illumination, worn lane markings, and asphalt irregularities. By applying classical image processing techniques from Gonzalez and Woods, the system aims to create reliable modules for lane tracking, traffic light classification, obstacele detection and barrier-state detection in unstructured environments.

1 Methods and Materials

1.1 Lane Detection and Tracking

The lane detection module employs a hybrid segmentation approach designed to function robustly under varying illumination conditions. The pipeline consists of three stages: feature extraction via morphological and chromatic fusion, geometric modeling using the Hough Transform, and temporal tracking.

1.1.1 Hybrid Feature Extraction

To isolate lane markings, we implement a logical disjunction of two masking techniques. First, a chromatic mask M_{color} is generated in the HSV space, targeting white, yellow, and a specific shadow-invariant subspace defined as $H \in [108, 130]$, $S \in [45, 52]$. Simultaneously, a morphological Top-Hat transform is applied to the grayscale image I_{gray} using a 19×19 rectangular kernel S to isolate high-contrast local structures:

$$I_{TH} = I_{gray} - (I_{gray} \circ S) \tag{1}$$

The final binary mask M_{final} is the union of the chromatic and morphological masks[2].

1.1.2 Geometric Extraction via Hough Transform

Edge detection is performed on M_{final} using the Canny operator. To detect linear lane boundaries from the resulting edge map, we utilize the **Probabilistic Hough Transform (PHT)**. Unlike the standard Cartesian representation ($y = mx + b$), the Hough Transform parameterizes lines in polar coordinates to avoid singularities with vertical lines: [4]

$$\rho = x \cos\theta + y \sin\theta \tag{2}$$

where ρ is the perpendicular distance from the origin to the line, and θ is the angle of the normal vector. Each edge pixel (x, y) in the image space votes for potential sinusoidal curves in the (ρ, θ) parameter space. The PHT optimization minimizes computational cost by analyzing a random subset of points to detect line segments defined by endpoints $P = \{(x_1, y_1), (x_2, y_2)\}$. Segments with slopes $|\frac{dy}{dx}| < 0.4$ are rejected as environmental noise.

1.1.3 Temporal Tracking and Stabilization

Detected line segments are organized into *Lane Candidates*. A candidate state $\mathbf{l}_t$ is updated using an Exponential Weighted Moving Average (EWMA) filter for temporal smoothness:

$$\mathbf{l}_t = \alpha \cdot \mathbf{l}_{new} + (1 - \alpha) \cdot \mathbf{l}_{t-1} \tag{3}$$

with $\alpha = 0.7$. A hysteresis filter confirms a lane only after $N_{min} = 8$ consecutive detections and retains it in memory for $N_{miss} = 5$ frames during temporary occlusions.

1.2 Barrier Position Detection

To robustly distinguish barrier states, we employ a sensor-fusion approach that validates geometric structure with chromatic signatures, effectively filtering background noise.

1.2.1 Geometric Structure Extraction

Vertical gradients are extracted from the grayscale ROI, I_{gray}, to isolate horizontal edges. The gradient G_y is computed via convolution with the Sobel-Y kernel K_y:

$$G_y = I_{gray} * K_y \quad \text{where} \quad K_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \tag{4}$$

The gradient magnitude is thresholded to form an edge map. To reconstruct the barrier's physical body from disjoint edges, we apply morphological dilation using an anisotropic rectangular structuring element S_{rect} (5×15 pixels):

$$M_{struct} = M_{edge} \oplus S_{rect} \tag{5}$$

1.2.2 Logic-Gated Verification

Simultaneously, chromatic masks M_{red} and M_{white} are generated in HSV space. To ensure spatial coherence, we calculate the pixel count of color features strictly *within* the structural mask:

$$N_{red} = |M_{red} \cap M_{struct}| \quad , \quad N_{white} = |M_{white} \cap M_{struct}| \tag{6}$$

The barrier is classified as **CLOSED** only if the geometric structure and both spectral components simultaneously exceed their respective thresholds τ:

$$State = \begin{cases} \text{CLOSED} & \text{if } (|M_{struct}| > \tau_S) \wedge (N_{red} > \tau_R) \wedge (N_{white} > \tau_W) \\ \text{OPEN} & \text{otherwise} \end{cases} \tag{7}$$

1.3 Traffic Light Detection and Classification

Traffic light detection is formulated as a constrained color-segmentation problem followed by geometric validation. Each frame $I(x, y)$ is mapped from RGB to HSV space, where chromatic components are less sensitive to illumination variations. A fixed Region of Interest (ROI) $\Omega_{\mathrm{TL}} \subset I$ is defined based on prior geometric knowledge of traffic light placement.

For each color class $k \in \{\mathrm{red}, \mathrm{yellow}, \mathrm{green}\}$, a binary mask is obtained via thresholding

$$M_k(x, y) = \begin{cases} 1, & \text{if } (H, S, V) \in \mathcal{T}_k \\ 0, & \text{otherwise} \end{cases}$$

where $\mathcal{T}_k$ denotes the HSV threshold set for class k. Morphological opening is applied to M_k to suppress small-scale noise. Connected components are extracted, and each candidate region $\mathcal{R}$ is evaluated using its area A, mean brightness [6]

$$\bar{V} = \frac{1}{|\mathcal{R}|} \sum_{(x,y) \in \mathcal{R}} V(x, y),$$

and circularity

$$C = \frac{4\pi A}{P^2},$$

where P is the contour perimeter. Frame-level decisions are temporally stabilized using majority voting over a finite window $\{t - N, \ldots, t\}$.

1.4 Obstacle Detection

Obstacle detection is treated as a structural anomaly detection problem using spatial edge statistics. After Gaussian smoothing, Canny edge detection is applied to obtain a binary edge map $E(x, y)$. Processing is restricted to a road-region ROI Ω_{obs}.

The edge map is partitioned into N vertical bins $\{B_i\}_{i=1}^{N}$, and column-wise edge density is computed as

$$d_i = \sum_{(x,y) \in B_i} E(x, y).$$

Bins satisfying

$$d_i > \mu_d + \alpha \sigma_d$$

are selected as candidates, where μ_d and σ_d denote the mean and standard deviation of $\{d_i\}$. For each candidate region, vertical localization is refined using row-wise edge accumulation

$$r(y) = \sum_{x} E(x, y),$$

yielding a bounding box constrained by geometric conditions on width, height, and aspect ratio. To enforce temporal consistency, detections are confirmed only if they persist across multiple frames using a sliding-window voting rule.

2 Results

2.1 Performance Evaluation Metrics

To rigorously assess the discrete classification tasks (barrier state and lane presence), we utilize standard statistical metrics derived from the confusion matrix elements: True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). The system's reliability is quantified using Accuracy, Precision, Recall, and the F1-Score, defined as follows:

$$\begin{aligned} \text{Accuracy} &= \frac{TP + TN}{TP + TN + FP + FN} \qquad & \text{Precision} &= \frac{TP}{TP + FP} \\ \text{Recall} &= \frac{TP}{TP + FN} \qquad & \text{F1-Score} &= 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \end{aligned} \tag{8}$$

These metrics provide a comprehensive view of the algorithmic performance, isolating geometric misclassifications from chromatic validation errors.

2.2 Lane Detection and Tracking Performance

The lane detection module, was evaluated for its performance in challenging conditions including varied lighting and shadows. While precise quantitative accuracy metrics are difficult to establish due to the continuous nature of lane presence and the inherent variability in road markings, the system consistently demonstrated robust lane identification. Qualitative results from diverse scenarios are presented in Figure 1. These images highlight the system's ability to detect and track lane boundaries effectively, even in the presence of significant illumination changes and partial occlusions.

(a) Detection in standard daylight

(b) Detection under varying light

Figure 1: Performance of the Hybrid Lane Detection system under varying environmental conditions.

2.3 Barrier State Detection Performance

The classification performance was evaluated across three annotated video sequences. On this test set, the system achieved a **Precision** of 100% and a **Recall** of 100%, resulting in a perfect

F1-Score of 1.0. These metrics quantitatively confirm that the logical fusion of geometric and chromatic features effectively eliminated False Positives (FP) caused by background clutter while maintaining maximum sensitivity to the barrier structure. Qualitative results are illustrated in Figure 2.

(a) Detected "Closed" State

(b) Detected "Open" State

Figure 2: Qualitative results of the Barrier Position Detection module on test data.

2.4 Traffic Light Detection and Classification Performance

The traffic signal recognition module was evaluated on a dataset containing 10 active traffic lights. The system demonstrated perfect sensitivity, successfully detecting all instances ($TP = 10$, $FN = 0$), yielding a **Recall of 100%**. This confirms the system's safety-critical reliability, ensuring no signals were missed.

However, the geometric filtering stage exhibited a tendency to over-segment, identifying 5 circular traffic signs as signal lights ($FP = 5$). Consequently, the **Precision** was calculated at 66.7%, resulting in an overall **F1-Score of 0.80**. Although the False Positive rate reduces precision, the design philosophy prioritizes Recall to absolutely eliminate the hazardous risk of failing to detect a stop signal. Detection results are visualized in Figure 3.

Red Light

Yellow Light

Green Light

False Positive

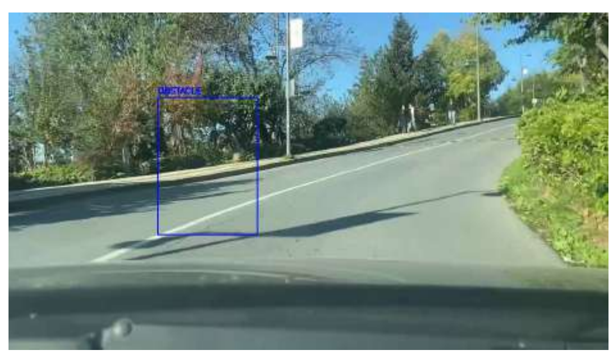
Figure 3: Representative traffic light detection results.

2.5 Obstacle Detection

The obstacle detection subsystem was evaluated on a test sequence yielding 60 total detection events. Analysis against ground truth data revealed 45 True Positives (TP) and 15 False Positives (FP), while 5 actual obstacles were missed ($FN = 5$).

This performance results in a **Precision of 75.0%**, indicating a moderate rate of false alarms caused by complex background textures. However, crucial for autonomous safety, the system achieved a high **Recall of 90.0%**, ensuring the majority of physical hazards were correctly identified. The overall algorithmic balance is reflected in an **F1-Score of 0.82**. Representative detection results are shown in Figure **??**.

(a) Correct detection: Motorcycle



(b) False Positive: Background noise

Figure 4: Qualitative evaluation of the Obstacle Detection module.

3 Discussion

The results highlight a clear performance dichotomy between structural and semantic tasks. The **Barrier** and **Lane** modules demonstrated superior robustness against environmental noise (shadows) compared to standard edge detection, evidenced by the barrier's perfect F1-Score (1.0). However, the **Traffic Light** and **Obstacle** modules revealed the deficiencies of classical geometric methods. While achieving high Recall to ensure safety, their lower Precision (66.7% and 75% respectively) indicates an inability to distinguish semantically similar objects (e.g., traffic signs vs. signals) without the learned features inherent to Deep Learning approaches, resulting in a higher rate of false alarms.

4 Conclusion

This project successfully validated a real-time classical perception stack while identifying key precision limitations. Future work will integrate a lightweight CNN to classify traffic light candidates, directly addressing the high false-positive rate noted in the discussion. Additionally, Optical Flow algorithms will be implemented to distinguish moving hazards from static background textures, improving obstacle detection precision. Finally, Inverse Perspective Mapping (IPM) will be adopted to linearize lane geometry for better curvature handling.

Bibliography

References

[1] Gonzalez, R. C., and Woods, R. E., *Digital Image Processing*, 4th ed., Pearson, 2018.

[2] Parajuli, A., Çelenk, M., and Riley, H. B., "Robust Lane Detection in Shadows and Low Illumination Conditions using Local Gradient Features," *Open Journal of Applied Sciences*, vol. 3, pp. 68–74, 2013.

[3] Du, M., Wang, J., Li, N., and Li, D., "Shadow Lane Robust Detection by Image Signal Local Reconstruction," *IJSPIPR*, vol. 9, no. 3, pp. 89–102, 2016.

[4] Hough, P. V. C., "Method and Means for Recognizing Complex Patterns," U.S. Patent 3,069,654, 1962.

[5] Duda, R. O., and Hart, P. E., "Use of the Hough Transformation to Detect Lines and Curves in Pictures," *Communications of the ACM*, vol. 15, no. 1, pp. 11–15, 1972.

[6] Omachi, M., and Omachi, S., "Traffic Light Detection with Color and Edge Information," Proc. *IEEE CIMSA*, pp. 284–287, 2009.

[7] Chae, S., Kim, S., and Pan, S., "Traffic Light Detection Algorithm based on Color and Shape Information," *Journal of the KITE*, vol. 41, pp. 35–42, 2004.

[8] Soille, P., *Morphological Image Analysis*, Springer, 2003.

[9] Bai, X., and Zhou, F., "Analysis of New Top-Hat Transformation and the Application for Infrared Small Target Detection," *Pattern Recognition Letters*, vol. 31, no. 14, pp. 2143–2152, 2010.

Line_Detector.py

```python
import cv2
import numpy as np

# --- SETTINGS ---
MIN_CONSECUTIVE_FRAMES = 8
MAX_MISSED_FRAMES = 5
MAX_DISTANCE_THRESHOLD = 50

# --- EXTRA: MORPHOLOGICAL OPERATION SETTINGS (Shadow Shape Detection) ---
MORPH_KERNEL_SIZE = 19
MORPH_THRESHOLD = 20

class LaneCandidate:
    """
    Class representing each potential lane segment.
    Tracks its own history, position, and reliability.
    """
    def __init__(self, line, img_height):
        self.img_height = img_height
        self.line = line
        self.found_count = 1
        self.missed_count = 0
        self.is_confirmed = False
        self.update_params(line)

    def update_params(self, line):
        x1, y1, x2, y2 = line
        if x2 == x1: return
        self.line = line

    def is_similar_to(self, new_line):
        ox1, oy1, ox2, oy2 = self.line
        nx1, ny1, nx2, ny2 = new_line

        dist1 = np.sqrt((ox1 - nx1)**2 + (oy1 - ny1)**2)
        dist2 = np.sqrt((ox2 - nx2)**2 + (oy2 - ny2)**2)

        if dist1 < MAX_DISTANCE_THRESHOLD and dist2 < MAX_DISTANCE_THRESHOLD:
            return True
        return False

    def update(self, new_line):
        self.found_count += 1
```

```python
        self.missed_count = 0

        # Smoothing (70% New, 30% Old)
        self.line = [
            int(0.7 * new_line[0] + 0.3 * self.line[0]),
            int(0.7 * new_line[1] + 0.3 * self.line[1]),
            int(0.7 * new_line[2] + 0.3 * self.line[2]),
            int(0.7 * new_line[3] + 0.3 * self.line[3])
        ]

        if self.found_count >= MIN_CONSECUTIVE_FRAMES:
            self.is_confirmed = True

    def mark_missing(self):
        self.missed_count += 1
        if self.missed_count > MAX_MISSED_FRAMES:
            return False
        return True

# Global candidate list
lane_candidates = []

def get_extended_line(line, img_height):
    x1, y1, x2, y2 = line
    if x2 == x1: return line

    poly = np.polyfit([y1, y2], [x1, x2], 1)

    y_bottom = img_height
    y_top = int(img_height * 0.6)

    x_bottom = int(poly[0] * y_bottom + poly[1])
    x_top = int(poly[0] * y_top + poly[1])

    return [x_bottom, y_bottom, x_top, y_top]

def process_lanes_tracking(img, raw_lines):
    global lane_candidates
    height, width = img.shape[:2]

    if raw_lines is None:
        raw_lines = []

    # 1. Extend raw lines
    extended_lines = []
    for line in raw_lines:
        x1, y1, x2, y2 = line[0]
```

```
            if abs(x2 - x1) < 1e-6: continue
            slope = (y2 - y1) / (x2 - x1)
            if abs(slope) < 0.4: continue

            ext_line = get_extended_line([x1, y1, x2, y2], height)
            extended_lines.append(ext_line)

        # 2. Matching
        matched_candidate_indices = set()

        for ext_line in extended_lines:
            found_match = False
            for i, candidate in enumerate(lane_candidates):
                if candidate.is_similar_to(ext_line):
                    candidate.update(ext_line)
                    matched_candidate_indices.add(i)
                    found_match = True
                    break

            if not found_match:
                new_cand = LaneCandidate(ext_line, height)
                lane_candidates.append(new_cand)

        # 3. Cleanup
        candidates_to_keep = []
        for i, candidate in enumerate(lane_candidates):
            if i in matched_candidate_indices:
                candidates_to_keep.append(candidate)
            else:
                still_alive = candidate.mark_missing()
                if still_alive:
                    candidates_to_keep.append(candidate)

        lane_candidates = candidates_to_keep

        # 4. Selection and Drawing
        confirmed_lanes = [c for c in lane_candidates if c.is_confirmed]
        final_left = None
        final_right = None
        center_x = width // 2

        best_left_candidates = []
        best_right_candidates = []

        for cand in confirmed_lanes:
            x_bottom = cand.line[0]
            if x_bottom < center_x:
```

```
            best_left_candidates.append(cand)
        else:
            best_right_candidates.append(cand)

    if best_left_candidates:
        final_left = max(best_left_candidates, key=lambda c: c.found_count).line

    if best_right_candidates:
        final_right = max(best_right_candidates, key=lambda c: c.found_count).line

    line_image = np.zeros_like(img)

    if final_left is not None:
        cv2.line(line_image, (final_left[0], final_left[1]), (final_left[2], final_left[3]), (255, 0, 0), 10)

    if final_right is not None:
        cv2.line(line_image, (final_right[0], final_right[1]), (final_right[2], final_right[3]), (0, 0, 255), 10)

    if final_left is not None and final_right is not None:
         pts = np.array([
            (final_left[0], final_left[1]),
            (final_left[2], final_left[3]),
            (final_right[2], final_right[3]),
            (final_right[0], final_right[1])
        ], np.int32)
         cv2.fillPoly(line_image, [pts], (0, 255, 0))

    return line_image

def process_frame(frame):
    height, width = frame.shape[:2]

    # ---------------------------------------------------------
    # SECTION 1: COLOR MASKS (HLS/HSV)
    # ---------------------------------------------------------
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    # 1.a) White Mask (Standard)
    lower_white = np.array([0, 0, 200])
    upper_white = np.array([180, 50, 255])
    mask_white = cv2.inRange(hsv, lower_white, upper_white)

    # 1.b) Yellow Mask (Standard)
```

```
    lower_yellow = np.array([15, 100, 100])
    upper_yellow = np.array([35, 255, 255])
    mask_yellow = cv2.inRange(hsv, lower_yellow, upper_yellow)

    # 1.c) SHADOW MASK (VALUES YOU FOUND)
    # Values: H:108-130, S:45-52, V:187-197
    lower_shadow = np.array([108, 45, 187])
    upper_shadow = np.array([130, 52, 197])
    mask_shadow_color = cv2.inRange(hsv, lower_shadow, upper_shadow)

    # Combine color masks
    mask_color_combined = cv2.bitwise_or(mask_white, mask_yellow)
    mask_color_combined = cv2.bitwise_or(mask_color_combined,
       mask_shadow_color)

    # ------------------------------------------------------------
    # SECTION 2: MORPHOLOGICAL OPERATION (TOP-HAT)
    # For color-independent shape detection
    # ------------------------------------------------------------
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Kernel: 19x19
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (
       MORPH_KERNEL_SIZE, MORPH_KERNEL_SIZE))
    tophat_img = cv2.morphologyEx(gray, cv2.MORPH_TOPHAT, kernel)

    # Threshold: 20
    _, mask_morph = cv2.threshold(tophat_img, MORPH_THRESHOLD, 255,
       cv2.THRESH_BINARY)

    # ------------------------------------------------------------
    # SECTION 3: MERGING AND EDGE DETECTION
    # ------------------------------------------------------------

    # Color Mask OR Morphology Mask
    final_mask = cv2.bitwise_or(mask_color_combined, mask_morph)

    # Apply mask to the original image (keep only masked regions)
    masked_frame = cv2.bitwise_and(frame, frame, mask=final_mask)

    # Convert to grayscale again and apply Canny (following your
       original pipeline)
    masked_gray = cv2.cvtColor(masked_frame, cv2.COLOR_BGR2GRAY)

    # Light blur (for noise reduction)
    blur = cv2.GaussianBlur(masked_gray, (5, 5), 0)

```

```
    # Canny Edge Detection
    edges = cv2.Canny(blur, 50, 150)

    # -----------------------------------------------------------
    # SECTION 4: ROI AND HOUGH
    # -----------------------------------------------------------

    # Original ROI from your code
    roi_vertices = np.array([[
        (int(width * 0.05), int(height * 0.8)),
        (int(width * 0.2), int(height * 0.5)),
        (int(width * 0.9), int(height * 0.5)),
        (int(width * 0.95), int(height * 0.8))
    ]], dtype=np.int32)

    mask_roi = np.zeros_like(edges)
    cv2.fillPoly(mask_roi, roi_vertices, 255)
    masked_edges = cv2.bitwise_and(edges, mask_roi)

    lines = cv2.HoughLinesP(
        masked_edges,
        rho=1,
        theta=np.pi/180,
        threshold=20,
        minLineLength=20,
        maxLineGap=300
    )

    line_layer = process_lanes_tracking(frame, lines)

    # Combine layers
    result = cv2.addWeighted(frame, 1.0, line_layer, 0.4, 0)
    return result
```

Listing 1: Initial trial for line detection in the campus

Barrier_detector.py

```python
import cv2
import numpy as np

def barrier_detection_hybrid(video_path):
    cap = cv2.VideoCapture(video_path)

    if not cap.isOpened():
        print("Error: Could not open video file!")
        return

    # --- SETTINGS ---
    # 1. ROI (Region of Interest) - Where the barrier sits when closed
    # Read the first frame to get dimensions
    ret, frame = cap.read()
    if not ret: return
    h, w = frame.shape[:2]

    # Define ROI coordinates (Adjust these percentages based on your specific video)
    roi_y1, roi_y2 = int(h * 0.3), int(h * 0.4)
    roi_x1, roi_x2 = int(w * 0.20), int(w * 0.85)

    # 2. Threshold Values
    # Minimum number of pixels required to consider the feature "present"
    MIN_EDGE_PIXELS = 200    # Is there a long enough horizontal line?
    MIN_RED_PIXELS = 50      # Are there red reflectors inside that line?
    MIN_WHITE_PIXELS = 50    # Is there a white body inside that line?

    print("Hybrid Analysis Started...")
    print("Logic: Horizontal Edge && Red Color && White Color -> CLOSED")

    while True:
        ret, frame = cap.read()
        if not ret:
            # Loop video
            cap.set(cv2.CAP_PROP_POS_FRAMES, 0)
            continue

        # Crop ROI
```

```python
        roi = frame[roi_y1:roi_y2, roi_x1:roi_x2]

        # ----------------------------------------------------------
        # STEP 1: SOBEL Y (Find Horizontal Structure)
        # ----------------------------------------------------------
        gray = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)

        # Sobel Y derivative (Detects only horizontal changes)
        sobel_y = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
        sobel_y = np.absolute(sobel_y)

        # Normalize to 0-255 and convert to uint8
        sobel_8u = np.uint8(255 * sobel_y / np.max(sobel_y + 1e-9))

        # Threshold: Keep only strong horizontal edges
        _, edge_mask = cv2.threshold(sobel_8u, 50, 255, cv2.THRESH_BINARY)

        # Dilation:
        # Edges are thin. We dilate the mask to cover the "body" of the barrier
        # so we can check for colors inside it. Using a horizontal kernel.
        kernel_dilate = np.ones((5, 15), np.uint8)
        structure_mask = cv2.dilate(edge_mask, kernel_dilate, iterations=1)

        # Structure Score (How much horizontal object is there?)
        structure_score = cv2.countNonZero(structure_mask)

        # ----------------------------------------------------------
        # STEP 2: COLOR ANALYSIS (Red and White)
        # ----------------------------------------------------------
        hsv = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)

        # Red Mask (For Reflectors)
        # Red wraps around 0/180 in HSV, so we need two ranges.
        mask_r1 = cv2.inRange(hsv, np.array([0, 100, 50]), np.array([10, 255, 255]))
```

```
        mask_r2 = cv2.inRange(hsv, np.array([170, 100, 50]), np.array([180, 255, 255]))
        red_mask = cv2.bitwise_or(mask_r1, mask_r2)

        # White Mask (For Barrier Body)
        # Low Saturation, High Value (Value lowered slightly to catch shadows)
        white_mask = cv2.inRange(hsv, np.array([0, 0, 150]), np.array([180, 50, 255]))

        # -----------------------------------------------------------
        # STEP 3: FUSION
        # We look for colors ONLY inside the "Structure Mask".
        # This prevents background objects (red cars, white buildings) from triggering detection.
        # -----------------------------------------------------------

        # Red pixels overlapping with horizontal edges
        red_in_barrier = cv2.bitwise_and(red_mask, red_mask, mask=structure_mask)
        red_score = cv2.countNonZero(red_in_barrier)

        # White pixels overlapping with horizontal edges
        white_in_barrier = cv2.bitwise_and(white_mask, white_mask, mask=structure_mask)
        white_score = cv2.countNonZero(white_in_barrier)

        # -----------------------------------------------------------
        # STEP 4: DECISION MECHANISM
        # -----------------------------------------------------------

        # Condition: Is there Structure? AND Red? AND White?
        is_structure_ok = structure_score > MIN_EDGE_PIXELS
        is_red_ok = red_score > MIN_RED_PIXELS
        is_white_ok = white_score > MIN_WHITE_PIXELS

        if is_structure_ok and is_red_ok and is_white_ok:
            status = "CLOSED (BARRIER DETECTED)"
            color_status = (0, 0, 255) # Red
```

```python

            # Draw a bounding box around the detected structure
            contours, _ = cv2.findContours(structure_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
            for cnt in contours:
                if cv2.contourArea(cnt) > 500: # Ignore small noise
                    x, y, w, h = cv2.boundingRect(cnt)
                    # Translate ROI coordinates to Frame coordinates
                    cv2.rectangle(frame, (roi_x1 + x, roi_y1 + y), (roi_x1 + x + w, roi_y1 + y + h), (0, 0, 255), 2)
        else:
            status = "OPEN (PASSAGE FREE)"
            color_status = (0, 255, 0) # Green

        # ----------------------------------------------------------
        # VISUALIZATION PANEL
        # ----------------------------------------------------------

        # 1. Structure Mask (Visualized as Blue)
        vis_structure = cv2.cvtColor(structure_mask, cv2.COLOR_GRAY2BGR)
        vis_structure[:, :, 0] = 255 # Boost Blue channel
        vis_structure[:, :, 1] = 0
        vis_structure[:, :, 2] = 0

        # 2. Color Masks (Visualized as Red and White)
        vis_red = cv2.cvtColor(red_in_barrier, cv2.COLOR_GRAY2BGR)
        vis_red[:, :, 2] = 255 # Red only

        vis_white = cv2.cvtColor(white_in_barrier, cv2.COLOR_GRAY2BGR) # White stays white

        # Blend them all into one analysis image
        # Structure (Base) + Red + White
        analysis_view = cv2.addWeighted(vis_structure, 0.3, vis_red, 1.0, 0)
        analysis_view = cv2.addWeighted(analysis_view, 1.0, vis_white, 1.0, 0)

        # Add Debug Text to the small panel
        cv2.putText(analysis_view, f"Struct: {structure_score}", (5, 15), cv2.FONT_HERSHEY_SIMPLEX, 0.4, (255, 255, 255), 1)
```

```python
        cv2.putText(analysis_view, f"Red: {red_score}", (5, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.4, (0, 0, 255), 1)
        cv2.putText(analysis_view, f"White: {white_score}", (5, 45), cv2.FONT_HERSHEY_SIMPLEX, 0.4, (200, 200, 200), 1)

        # Resize panel for display
        analysis_view_large = cv2.resize(analysis_view, (400, 150))

        # Draw ROI Box and Status on Main Frame
        cv2.rectangle(frame, (roi_x1, roi_y1), (roi_x2, roi_y2), color_status, 2)
        cv2.putText(frame, status, (roi_x1, roi_y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.8, color_status, 2)

        # Attach Analysis Panel to the right side of the frame
        final_h = frame.shape[0]

        side_panel = np.zeros((final_h, 400, 3), dtype=np.uint8)
        # Center the analysis view vertically
        y_offset = (final_h - 150) // 2
        side_panel[y_offset:y_offset+150, :] = analysis_view_large

        # Explanatory Text on Side Panel
        cv2.putText(side_panel, "ANALYSIS DETAIL", (20, y_offset - 20), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 255), 2)
        cv2.putText(side_panel, "Blue: Sobel (Horizontal)", (20, y_offset + 170), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 1)
        cv2.putText(side_panel, "Red: Reflector", (20, y_offset + 190), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 1)
        cv2.putText(side_panel, "White: Body", (20, y_offset + 210), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (200, 200, 200), 1)

        final_frame = np.hstack((frame, side_panel))

        cv2.imshow("Hybrid Barrier Detection", final_frame)

        if cv2.waitKey(30) & 0xFF == 27: # ESC to exit
            break

    cap.release()
    cv2.destroyAllWindows()

# --- RUN ---
# Make sure to update the filename to your video path
barrier_detection_hybrid('bariyer.mp4')
```

Listing 2: Initial trial for line detection in the campus

Traffic_Light.py

```python
import cv2
import numpy as np
from collections import deque, Counter
from src.config import *

# ---------------------------------------------------
# Configuration
# ---------------------------------------------------
DEBUG_VISUALIZATION = False
VOTING_WINDOW = 4  # temporal window for state voting

# Buffer to store recent frame-level decisions
state_buffer = deque(maxlen=VOTING_WINDOW)

# ---------------------------------------------------
# Main traffic light detection function
# ---------------------------------------------------
def detect_and_classify_traffic_light(frame):
    h, w, _ = frame.shape

    # Define Region of Interest (upper-central area)
    y1, y2 = 0, int(0.3 * h)
    x1, x2 = int(0.2 * w), int(0.8 * w)

    roi = frame[y1:y2, x1:x2]
    hsv_roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)

    # Generate color masks in HSV space
    red_mask1 = cv2.inRange(hsv_roi, RED_LOWER_1, RED_UPPER_1)
    red_mask2 = cv2.inRange(hsv_roi, RED_LOWER_2, RED_UPPER_2)
    red_mask = cv2.bitwise_or(red_mask1, red_mask2)

    yellow_mask = cv2.inRange(hsv_roi, YELLOW_LOWER, YELLOW_UPPER)
    green_mask = cv2.inRange(hsv_roi, GREEN_LOWER, GREEN_UPPER)

    # Detect valid blobs for each color
    red_detected = _process_color(frame, hsv_roi, red_mask, "RED", x1, y1)
    yellow_detected = _process_color(frame, hsv_roi, yellow_mask, "YELLOW", x1, y1)
    green_detected = _process_color(frame, hsv_roi, green_mask, "GREEN", x1, y1)

    # Frame-level decision logic
    if red_detected:
```

```
            state = "RED"
        elif yellow_detected:
            state = "YELLOW"
        elif green_detected:
            state = "GREEN"
        else:
            state = "UNKNOWN"

        # Temporal majority voting for stability
        state_buffer.append(state)
        voted_state = Counter(state_buffer).most_common(1)[0][0]

        # Overlay detected state
        cv2.putText(
            frame,
            f"TL STATE: {voted_state}",
            (30, 40),
            cv2.FONT_HERSHEY_SIMPLEX,
            1.0,
            (255, 255, 255),
            2
        )
        return frame, voted_state
# --------------------------------------------------
# Color-specific blob validation
# --------------------------------------------------
def _process_color(frame, hsv_roi, mask, label, x_offset, y_offset):
    # Morphological opening to remove noise
    kernel = np.ones((5, 5), np.uint8)
    clean_mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)

    # Extract connected components
    contours, _ = cv2.findContours(
        clean_mask,
        cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE
    )
    detected = False

    for cnt in contours:
        area = cv2.contourArea(cnt)
        if area < 30:
            continue

        # Compute mean HSV values inside contour
        contour_mask = np.zeros(clean_mask.shape, dtype=np.uint8)
        cv2.drawContours(contour_mask, [cnt], -1, 255, -1)
```

```python
        _, _, mean_v, _ = cv2.mean(hsv_roi, mask=contour_mask)

        # Shape descriptor: circularity
        perimeter = cv2.arcLength(cnt, True)
        circularity = 4 * np.pi * area / (perimeter ** 2 + 1e-6)

        # Validation criteria
        pass_area = 30 < area < 900
        pass_brightness = mean_v > 120
        pass_circularity = circularity > 0.75

        passed = pass_area and pass_brightness and pass_circularity

        # Optional visualization
        if DEBUG_VISUALIZATION or passed:
            box_color = (
                (0, 0, 255) if label == "RED" else
                (0, 255, 255) if label == "YELLOW" else
                (0, 255, 0)
            )

            x, y, w, h = cv2.boundingRect(cnt)
            cv2.rectangle(
                frame,
                (x + x_offset, y + y_offset),
                (x + w + x_offset, y + h + y_offset),
                box_color,
                2
            )

            if passed:
                cv2.putText(
                    frame,
                    label,
                    (x + x_offset, y + y_offset - 8),
                    cv2.FONT_HERSHEY_SIMPLEX,
                    0.6,
                    box_color,
                    2
                )
        if passed:
            detected = True

    return detected
```

Listing 3: Initial trial for line detection in the campus

Obstacle_detection.py

```
import cv2
import numpy as np
from collections import deque

DEBUG_VISUALIZATION = False

# -------------------------------------------------
# Temporal voting configuration
# -------------------------------------------------
VOTING_WINDOW = 5
VOTING_THRESHOLD = 3
obstacle_history = deque(maxlen=VOTING_WINDOW)


# -------------------------------------------------
# Region of Interest definition
# -------------------------------------------------
def get_roi_mask(frame):
    h, w = frame.shape[:2]

    roi_pts = np.array([[
        (int(0.25 * w), int(0.7 * h)),
        (int(0.75 * w), int(0.7 * h)),
        (int(0.75 * w), int(0.25 * h)),
        (int(0.25 * w), int(0.25 * h))
    ]], dtype=np.int32)

    mask = np.zeros((h, w), dtype=np.uint8)
    cv2.fillPoly(mask, roi_pts, 255)

    return mask, roi_pts


# -------------------------------------------------
# Main obstacle detection function
# -------------------------------------------------
def detect_obstacles(frame):
    h, w = frame.shape[:2]
    output = frame.copy()

    # ---------------- ROI ----------------
    roi_mask, roi_pts = get_roi_mask(frame)

    # ---------------- Preprocessing ----------------
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

```python
    blur = cv2.GaussianBlur(gray, (5, 5), 0)

    edges = cv2.Canny(blur, 60, 160)
    edges = cv2.bitwise_and(edges, edges, mask=roi_mask)
    edges = cv2.dilate(edges, np.ones((3, 3), np.uint8), iterations
       =1)

    # ---------------- Vertical Edge Density ----------------
    num_bins = 24
    bin_width = w // num_bins
    density = np.zeros(num_bins)

    for i in range(num_bins):
        x1 = i * bin_width
        x2 = (i + 1) * bin_width
        density[i] = cv2.countNonZero(edges[:, x1:x2])

    mean_d = np.mean(density)
    std_d = np.std(density)
    active_bins = density > (mean_d + 1.2 * std_d)

    # ---------------- Group Adjacent Active Bins ----------------
    candidates = []
    visited = np.zeros(num_bins, dtype=bool)

    for i in range(num_bins):
        if not active_bins[i] or visited[i]:
            continue

        left, right = i, i
        while left > 0 and active_bins[left - 1]:
            left -= 1
        while right < num_bins - 1 and active_bins[right + 1]:
            right += 1

        visited[left:right + 1] = True

        x1 = left * bin_width
        x2 = (right + 1) * bin_width
        bw = x2 - x1

        band = edges[:, x1:x2]

        # ---------------- Vertical Localization ----------------
        row_density = np.sum(band > 0, axis=1).astype(np.float32)

        row_density = cv2.GaussianBlur(
```

```
            row_density.reshape(-1, 1),
            (1, 21),
            0
        ).flatten()

        rd_mean = np.mean(row_density)
        rd_std = np.std(row_density)
        active_rows = row_density > (rd_mean + 1.0 * rd_std)

        if np.count_nonzero(active_rows) < 30:
            continue

        ys = np.where(active_rows)[0]
        y1 = int(ys[0])
        y2 = int(ys[-1])
        bh = y2 - y1

        # ---------------- Geometric Validation ----------------
        pass_width = bw > 0.08 * w
        pass_height = bh > 0.12 * h
        pass_bottom = y2 > 0.6 * h
        pass_aspect = bh / (bw + 1e-6) > 0.6

        density_score = np.sum(active_rows) / (bh + 1e-6)
        pass_density = density_score > 0.15

        passed = (
            pass_width and
            pass_height and
            pass_bottom and
            pass_aspect and
            pass_density
        )

        if passed:
            candidates.append((x1, y1, bw, bh))

    # ---------------- Temporal Voting ----------------
    obstacle_history.append(candidates)

    confirmed = []
    for bx, by, bw, bh in candidates:
        votes = 0
        for past in obstacle_history:
            for px, py, pw, ph in past:
                if (
                    min(bx + bw, px + pw) > max(bx, px) and
```

```
                    min(by + bh, py + ph) > max(by, py)
                ):
                    votes += 1
                    break

        if votes >= VOTING_THRESHOLD:
            confirmed.append((bx, by, bw, bh))

    # ---------------- Visualization ----------------
    for (x, y, bw, bh) in confirmed:
        cv2.rectangle(output, (x, y), (x + bw, y + bh), (255, 0, 0),
            2)
        cv2.putText(
            output,
            "OBSTACLE",
            (x, y - 8),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.6,
            (255, 0, 0),
            2
        )

    if DEBUG_VISUALIZATION:
        overlay = output.copy()
        cv2.polylines(overlay, roi_pts, True, (0, 255, 255), 2)
        cv2.addWeighted(overlay, 0.4, output, 0.6, 0, output)
        cv2.imshow("Edges (ROI)", edges)

    return output, confirmed
```

Listing 4: Initial trial for line detection in the campus