Road Lane Detection 2

Tuesday, September 5, 2023 1:17 PM

We start by declaring 2 lists that are used to store centroids (we are going to calculate them later). of current frame. So that we can use these values if no centroids are detected.

```
previous_frame_leftLine_centroids = []
previous_frame_rightLine_centroids = []
```

We start reading the video file. If we are successfully able to read the file, then we go inside while loop.

```
cap = cv2.VideoCapture('./data/test_video.mp4')
success, frame = cap.read()
```

Inside while loop:

step 1:

applyHLS() : it is a custom function that we defined. This function will take "current frame" as input

it converts frame into HLS from BGR. We will create a colored mask, which allows specified range of colors to be visible.

Then we use bitwise_and operation on original frame and color_mask.

```
def applyHLS(img):
    hls_img = cv2.cvtColor(img, cv2.COLOR_BGR2HLS)
    white_lower = np.array([0, 170, 0])    # white can be obtained from all the colors with
#Value (lightness) in range of 170, 255
    white_upper = np.array([255, 255, 255])
    yellow_lower = np.array([10, 0, 100])
    yellow_upper = np.array([50, 255, 255])
    white_mask = cv2.inRange(hls_img, white_lower, white_upper)
    yellow_mask = cv2.inRange(hls_img, yellow_lower, yellow_upper)
    colored_mask = cv2.bitwise_or(yellow_mask, white_mask)
    colored_mask = cv2.cvtColor(colored_mask, cv2.COLOR_GRAY2BGR)
    color_masked_frame = cv2.bitwise_and(img, colored_mask)
    return color_masked_frame
```

```
color_masked_frame = applyHLS(frame)
```

Step 2: Apply Gaussian Blur to smoothen the image.

```
blurreded_frame = cv2.GaussianBlur(color_masked_frame, (3, 3), cv2.BORDER_DEFAULT)
```

Step 3: select pixel coordinates for Region Of Interest.

```
rect = [[450, 400], [770, 400], [1250, 720], [200, 720]]
```

Step 4: transform the selected ROI into Bird's Eye View using custom defined birdsEyeView() function.

```
transformed_img = birdsEyeView(blurreded_frame, rect)
```

birdsEyeView() : it takes an img, rect as input.

We calculate width height of new image (i.e bird's eye view of selected ROI).

birdsEyeView() : it takes an img, rect as input.

We calculate width, height of new image (i.e bird's eye view of selected ROI).

Formula used : dist b/w (x_1, y_1) & (x_2, y_2) = $\sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}$

We use maxWidth, maxHeight as dimentions of BirdsEyeView image, where,

maxWidth = maximum of distance b/w Topright & Top left pt's of ROI and
dist b/w Bottomright & Bottomleft pt's of ROI.

maxHeight = maximum of distance b/w Bottom-Right & Top-right pt's of ROI and
" " Bottom Left & Top left " "

We define coordinates for transformed img (Bird's Eye View)

We calculate perspective transformation matrix using cv2.getPerspectiveTransform()
Then, we transform the ROI into a BirdsEyeView and return the image.

```
def birdsEyeView(img, rect):    # rect = [roiTopLeft, roiTopRight, roiBottomRight, roiBottomLeft]
    roiTopLeft, roiTopRight, roiBottomRight, roiBottomLeft = rect
    roiCornerPointsArray = np.float32([roiTopLeft, roiTopRight, roiBottomRight, roiBottomLeft])
    # We are calculating the width of new image = max (distange between bottomright, bottomleft. and dist 
between topright, topleft)
    width1 = np.sqrt(((roiBottomRight[0]-roiBottomLeft[0])**2) + ((roiBottomRight[1]-roiBottomLeft[1])**
2))
    width2 = np.sqrt(((roiTopRight[0]-roiTopLeft[0])**2) + ((roiTopRight[1]-roiTopLeft[1])**2))
    maxWidth = max(int(width1), int(width2))
    # We are calculating the height of new image = max (distange between bottomright, topright. and dist 
between bottomleft, topleft)
    height1 = np.sqrt(((roiBottomRight[0]-roiTopRight[0])**2) + ((roiBottomRight[1]-roiTopRight[1])**2))
    height2 = np.sqrt(((roiBottomLeft[0]-roiTopLeft[0])**2) + ((roiBottomLeft[1]-roiTopLeft[1])**2))
    maxHeight = max(int(height1), int(height2))

    # Dst img coordinates (src is transformed to dst)
    dstTopLeft = [0, 0]
    dstTopRight = [maxWidth-1, 0]
    dstBottomRight = [maxWidth-1, maxHeight-1]
    dstBottomLeft = [0, maxHeight-1]
    dstImgParams = np.float32([dstTopLeft, dstTopRight, dstBottomRight, dstBottomLeft])
    # Compute Perspective transformation matrix
    matrix = cv2.getPerspectiveTransform(roiCornerPointsArray, dstImgParams)
    # Warp input according to the matrix
    transformed_img = cv2.warpPerspective(img, matrix, (maxWidth, maxHeight))
    return transformed_img
```

Step 5: convert transformed_img into grayscale.

```
gray_transformed_img = cv2.cvtColor(transformed_img, cv2.COLOR_BGR2GRAY)
```

Step 6: Apply binary threshold on gray image.

```
ret, thresh = cv2.threshold(gray_transformed_img, 50, 255, cv2.THRESH_BINARY)
```

Step 7: Calculate centroids for both left line and Right line using custom function applySlidingWindow().

```
leftLine_centroids, rightLine_centroids = applySlidingWindow(thresh)
```

applySlidingWindow() : it takes thresholded image as input.

```
leftLine_centroids, rightLine_centroids = applySlidingWindow(thresh)
```

applySlidingWindow() : it takes thresholded image as input.

calculate histogram: it gives us distribution of white pixels.

we define 3 variables:

midpoint : it is half the width of transformed img (= thresh)

left_base : x-coordinate of pixel where whiteness is max of 1st half

right_base : x- " " " in 2nd half

We are going to start sliding windows process from height y = 560 (even though img height is 567 because if we start from 560, then we can equally divide the height into 14 sliding windows with out fraction part).

we took y = 560 just for simplicity in calculation.

For every sliding window we calculate contours and then moments.
with help of moments, we can then calculate centroids

we do this process for Left, Right lines and store all centroids

we store those centroids in global var that we defined at beginning.
so that if we don't find a single centroid in entire frame, we can use the centroids stored in global variables.

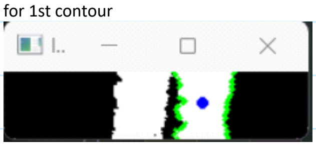
```
def applySlidingWindow(thresh):
    # Calculating Histogram.
    hist = np.sum(thresh[thresh.shape[0]//2:,:], axis=0)
    midpoint = int(hist.shape[0]/2)  #thresh img width/2 . We are dividing full image width into 2 parts.
    left_base = np.argmax(hist[:midpoint]) # x-coord of maximum peak of white pixel in left side or thresh
img
    right_base = np.argmax(hist[midpoint:]) + midpoint   # x-coord of maximum peak of white pixel in right
side or thresh img
    # Code for sliding window. All the bwlow values are calculated based on no.of sliding windows for each
frame. We have decided to take 14 windows.
    y = 560
    leftLine_centroids = [] # to store the x-coord of centroid for every contour detected in a sliding
window of Left Line
    rightLine_centroids = []
    global previous_frame_leftLine_centroids, previous_frame_rightLine_centroids  # used to store centroid
coords to be used later incase there is no centroid detected in current frame.
    window_count = 1
    while y > 0:
        # For Left Line
        left_line_roi = thresh[y-40:y, left_base-90:left_base+90]
        # we find contours (curve joining all continious points) of the left_line_roi
        contours, _ = cv2.findContours(left_line_roi, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
        if len(contours) > 0:
            _cx, _cy = [], []  # these list's are used to store all the centroids of detected contours in
a sliding window
            for index, contour in enumerate(contours):
                M = cv2.moments(contour)   # moments returns a dict of image moments. They can be used to
find useful infor like contour area, centroid, .....
                if M['m00'] != 0:
                    cx = int(M['m10'] / M['m00'])   # this gives us x-coord of centroid in pixels w.r.to
sliding dindow dimentions
                    cy = int(M['m01'] / M['m00'])   # this gives us y-coord of centroid in pixels w.r.to
sliding dindow dimentions
                    # We are converting cx, cy values w.r.to thresh i.e Bird's eye view image and
appending them to _cx, _cy
                    _cx.append(left_base-90+cx)
                    _cy.append(y-cy)
```

```
sliding dindow dimentions
                    # We are converting cx, cy values w.r.to thresh i.e Bird's eye view image and
appending them to _cx, _cy
                    _cx.append(left_base-90+cx)
                    _cy.append(y-cy)
            _cx = np.array(_cx)
            _cy = np.array(_cy)
            # Remove any NaN values from the _cx array
            _cx = _cx[~np.isnan(_cx)]
            _cy = _cy[~np.isnan(_cy)]
            if len(_cx)>0 and len(_cy)>0:
                mean_cx = int(np.mean(_cx))
                mean_cy = int(np.mean(_cy))
                leftLine_centroids.append((mean_cx, mean_cy))

        # For Right Lane.
        right_line_roi = thresh[y-40:y, right_base-90:right_base+90]
        # we find contours (curve joining all continious points) of the right_line_roi
        contours, _ = cv2.findContours(right_line_roi, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
        if len(contours) > 0:
            _cx, _cy = [], []  # these list's are used to store all the centroids of detected contours in
a sliding window
            for index, contour in enumerate(contours):
                M = cv2.moments(contour)   # moments returns a dict of image moments. They can be used to
find useful infor like contour area, centroid, .....
                if M['m00'] != 0:
                    cx = int(M['m10'] / M['m00'])   # this gives us x-coord of centroid in pixels w.r.to
sliding dindow dimentions
                    cy = int(M['m01'] / M['m00'])   # this gives us y-coord of centroid in pixels w.r.to
sliding dindow dimentions
                    # We are converting cx, cy values w.r.to thresh i.e Bird's eye view image and
appending them to _cx, _cy
                    _cx.append(right_base-90+cx)
                    _cy.append(y-cy)
            _cx = np.array(_cx)
            _cy = np.array(_cy)
            # Remove any NaN values from the _cx array
            _cx = _cx[~np.isnan(_cx)]
            _cy = _cy[~np.isnan(_cy)]
            if len(_cx)>0 and len(_cy)>0:
                mean_cx = int(np.mean(_cx))
                mean_cy = int(np.mean(_cy))
                rightLine_centroids.append((mean_cx, mean_cy))
        y -= 40
        window_count += 1
    if len(leftLine_centroids) > 1:
        previous_frame_leftLine_centroids = leftLine_centroids
    else:
        leftLine_centroids = previous_frame_leftLine_centroids

    if len(rightLine_centroids) > 1:
        previous_frame_rightLine_centroids = rightLine_centroids
    else:
        rightLine_centroids = previous_frame_rightLine_centroids

    return [leftLine_centroids, rightLine_centroids]
```

1st window

we got 3 contours.

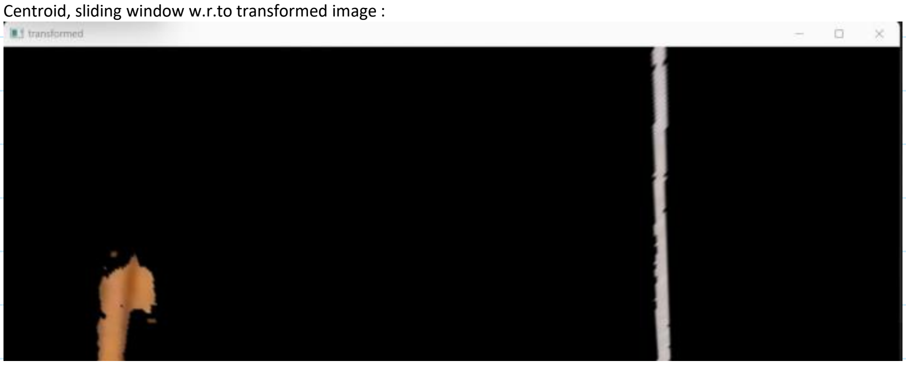
for 1st contour



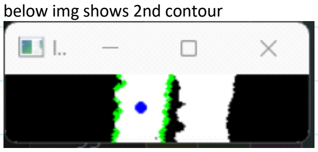
green lines --> contour
blue dot ---> centroid (calculated using cv2.moments)

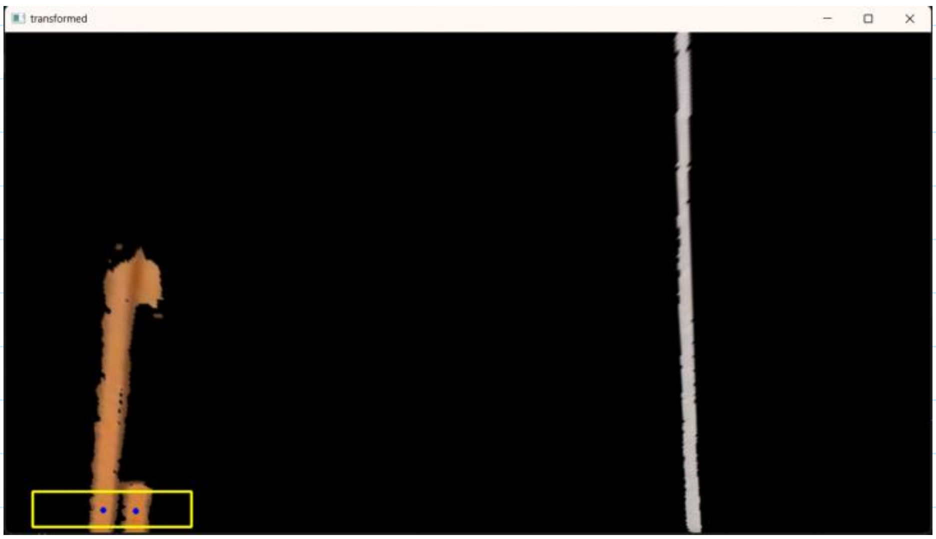
Above pic and dimentions are w.r.to 1st sliding window

Centroid, sliding window w.r.to transformed image :

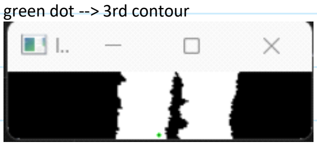


for 2nd contour :
below img shows 2nd contour

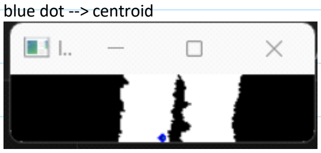


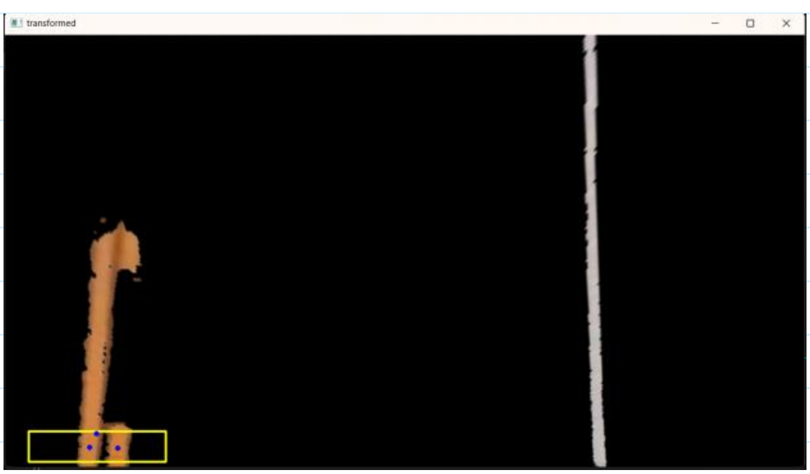


for 3rd contour :
green dot --> 3rd contour

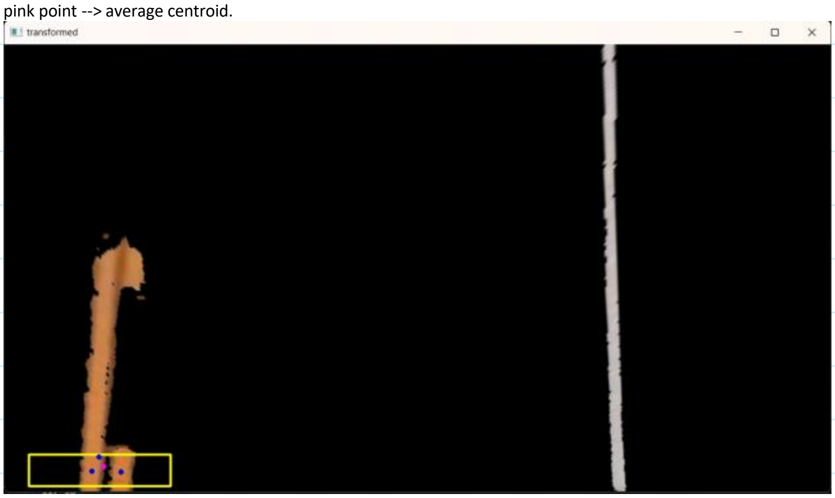


blue dot --> centroid





We calculate mean of all the 3 centroids and get a single point.
pink point --> average centroid.



Now, we use this average centroid's X coordinate as a left_base for 2nd sliding window

We keep on update the left_base value based on previous sliding window's average Centroid.

. at the end of the process for 1st frame, we will be having no.of centroids between 0 & 14.

if we have 0 or 1 centroids in current frame, then we take centroid values from previous frame and draw the curve.

Step 8: create a transformed_img_mask so that we can draw detected lines and region to drive.

```
transformed_img_mask = np.zeros_like(transformed_img)
```

Step 9, 10: Draw Road lines using applyDrawLeftLine()
applyDrawRightLine()

```
transformed_img_mask = applyDrawLeftLine(transformed_img_mask, leftLine_centroids, polygon_coords)
transformed_img_mask = applyDrawRightLine(transformed_img_mask, rightLine_centroids, polygon_coords)
```

```
def applyDrawLeftLine(transformed_img_mask, centroids, polygon_coords):
    h, w = transformed_img_mask.shape[:2]
    pts = []
    """ for x in range(w):
        y = np.polyval(coeffs, x)
        y = int(round(y))
        pts.append((x, y))
    pts = np.array(pts)
    mean_x = int(np.mean([centroid[0] for centroid in centroids]))
    cv2.polylines(transformed_img, [pts], False, (0, 0, 255), 2) """
    min_x = int(np.min([centroid[0] for centroid in centroids]))
    max_x = int(np.max([centroid[0] for centroid in centroids]))
    polygon_coords.append((min_x, h))
    polygon_coords.append((max_x, 0))
    cv2.line(transformed_img_mask, (max_x, 0), (min_x, h), (255, 0, 0), 15, 1)

    return transformed_img_mask
def applyDrawRightLine(transformed_img_mask, centroids, polygon_coords):
```

```
        polygon_coords.append((max_x, 0))
        cv2.line(transformed_img_mask, (max_x, 0), (min_x, h), (255, 0, 0), 15, 1)

        return transformed_img_mask
    def applyDrawRightLine(transformed_img_mask, centroids, polygon_coords):
        h, w = transformed_img_mask.shape[:2]
        pts = []
        """ for x in range(w):
            y = np.polyval(coeffs, x)
            y = int(round(y))
            pts.append((x, y))
        pts = np.array(pts)
        mean_x = int(np.mean([centroid[0] for centroid in centroids]))
        cv2.polylines(transformed_img, [pts], False, (0, 0, 255), 2) """
        min_x = int(np.min([centroid[0] for centroid in centroids]))
        max_x = int(np.max([centroid[0] for centroid in centroids]))
        polygon_coords.append((min_x, 0))
        polygon_coords.append((max_x, h))
        cv2.line(transformed_img_mask, (min_x, 0), (max_x, h), (255, 0, 0), 15, 1)

        return transformed_img_mask
```

step 11: Visualize all centroids detected

```
for centroid in leftLine_centroids:
    cv2.circle(transformed_img_mask, (centroid[0], centroid[1]), 2, (0, 0, 255), 5, 1)
for centroid in rightLine_centroids:
    cv2.circle(transformed_img_mask, (centroid[0], centroid[1]), 2, (0, 0, 255), 5, 1)
```

step 12: Draw optimal detected driving region using applyDrawDrivingRegion()

```
transformed_img_mask = applyDrawDrivingRegion(transformed_img_mask, polygon_coords)
```

```
def applyDrawDrivingRegion(transformed_img_mask, polygon_coords):
    #print(polygon_coords)
    # Convert the polygon_coords list to a numpy array
    polygon_coords = np.array(polygon_coords)
    # Reshape the array to have shape (n, 1, 2)
    polygon_coords = polygon_coords.reshape((-1, 1, 2))
    #print(polygon_coords)
    cv2.fillPoly(transformed_img_mask, [polygon_coords], color=(0, 255, 127))
    return transformed_img_mask
```

step 13: Project back the transformed_img mask back on to original frame. using applyBirdsEyeViewBackOnFrame()
Process is similar to applyBirdsEyeView()

```
transformed_frame = applyBirdsEyeViewBackOnFrame(frame, rect, transformed_img_mask)
```

```
def applyBirdsEyeViewBackOnFrame(img, rect, transformed_img_mask):
    roiTopLeft, roiTopRight, roiBottomRight, roiBottomLeft = rect
    roiCornerPointsArray = np.float32([roiTopLeft, roiTopRight, roiBottomRight, roiBottomLeft])
    maxHeight, maxWidth = transformed_img_mask.shape[:2]
    original_frame_h , original_frame_w = img.shape[:2]

    # Dst img coordinates (src is transformed to dst)
    dstTopLeft = [0, 0]
    dstTopRight = [maxWidth, 0]
    dstBottomRight = [maxWidth, maxHeight]
    dstBottomLeft = [0, maxHeight]
    dstImgParams = np.float32([dstTopLeft, dstTopRight, dstBottomRight, dstBottomLeft])
    # Compute Perspective transformation matrix
    inversematrix = cv2.getPerspectiveTransform(dstImgParams, roiCornerPointsArray)
    # Warp input according to the matrix
    projected_frame = cv2.warpPerspective(transformed_img_mask, inversematrix, (original_frame_w,
original_frame_h))
    # Blend the projected ROI with the original frame using alpha blending
    alpha = 0.35 # Adjust this value to change the transparency of the projected ROI
```

```
blended_frame = cv2.addWeighted(img, 1-alpha, projected_frame, alpha, 0)
return blended_frame
```