Aaron Iglesias
Udacity
Self-Driving Car Engineer Nanodegree
Advanced Lane Finding
03/01/2017

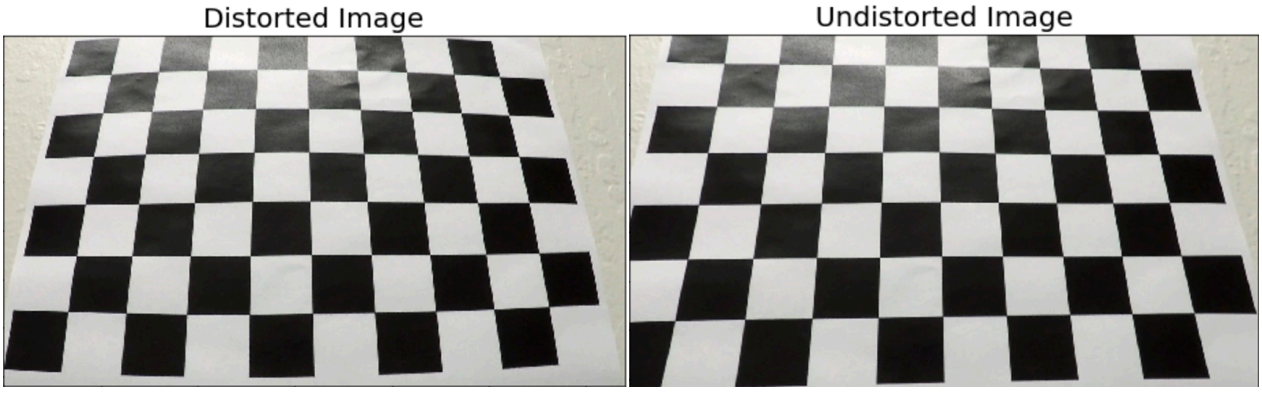
Camera Calibration

Briefly state how you computed the camera matrix and distortion coefficients. Provide an example of a distortion corrected calibration image.

The first step is to find the object points and image points by detecting corners from a series of distorted chessboard images. The object points represent the corners of an undistorted chess board, whereas the image points represent the corners of a distorted chessboard. The object points always remain the same, whereas the image points are calculated using OpenCV's findChessboardCorners() function. For this exercise, the number of corners for a given row is expected to be 9 and the number of corners in a given column is expected to be 6. Whenever findChessboardCorners() finds the expected number of corners in a distorted image, we record another pair of object points and detected image points.

The second step is to pass these object points, image points, and image size into OpenCV's calibrateCamera() function. This returns the desired camera matrix and distortion coefficients used to un-distort a given image taken with the same camera.

Here is an example of a distortion corrected calibration image:

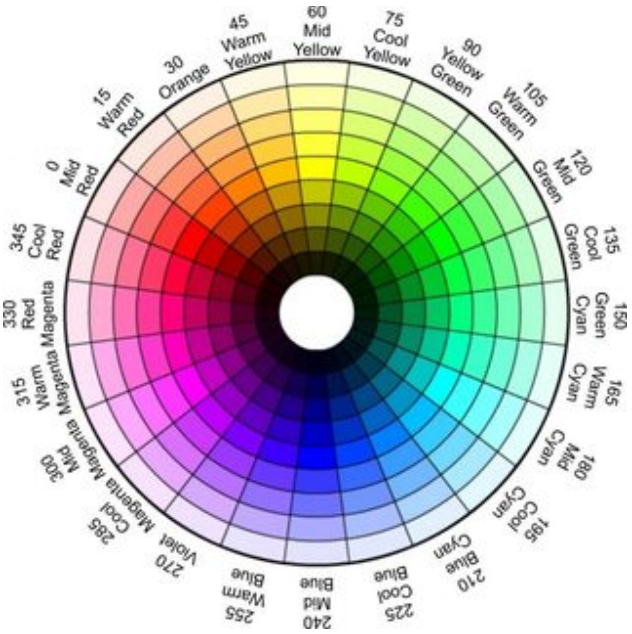


Pipeline (test images)

Provide an example of a distortion-corrected image.

Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

HSL (hue, saturation. lightness) color transform was sufficient for finding the lane lines. I referred to the following color wheel for hue (H) values:



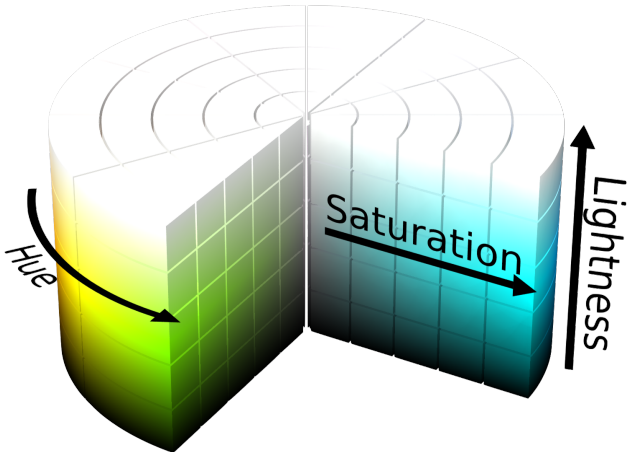
As we can see, values for H range from 0 to 360. However, in OpenCV, these values range from 0 to 180.

First, I detected yellow lanes. For yellow lanes, values may range from the lower end of orange to the upper end of mid-yellow. Referring to the wheel, these values range from 22.5 to 67.5. In OpenCV, these values are halved so non-yellow can be filtered out with H values in the range [11, 33].

However, hue alone is not enough for finding yellow lanes because the bridge in the project video also falls within the same hue range. The difference between the yellow lanes and the bridge is their saturation (S) levels with yellow lanes being more saturated. So, through trial and error, I found that S values in the range [86, 255] effectively filtered out the bridge while leaving the yellow lanes intact.

Lightness (L) is also needed to filter out shadows. Through trial and error, I found that L values in the range [20, 255] effectively filtered out shadows.

Second, I detected white lanes. For this, I referred to the following HSL image:



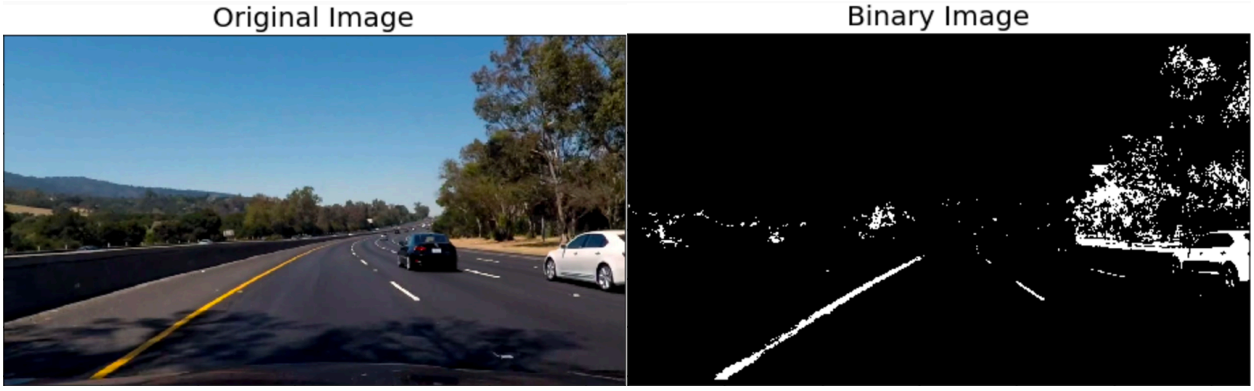
As we can see, white is independent of both hue and saturation, depending only on lightness. This was easy and, through trial and error, I found that L in the range [198, 255] effectively detected white lanes.

Third, I thresholded the HSL-transformed image to create two binary images: one for yellow lanes and one for white lanes. Note that I created two binary images because it is helpful to know whether a given lane is yellow or white. For yellow lanes, I thresholded the image with H in range [11, 33], S in range [86, 255], and L in range [20, 255]. For white lanes, I thresholded the image with L in range [198, 255].

Last, I combined these two binary images using cv2's bitwise_or() operation.

Here is my code (located in the Color/Gradient Threshold section of my Jupyter Notebook) along with an example of a binary image result:

```
def lane_binary(rgb_img):
    hls = cv2.cvtColor(rgb_img, cv2.COLOR_RGB2HLS)
    H = hls[:,:,0]
    L = hls[:,:,1]
    S = hls[:,:,2]
    yellow, white = np.zeros_like(H), np.zeros_like(H)
    yellow[(H >= 11) & (H <= 33) & (S >= 86) & (L >= 20)] = 1
    white[(L >= 198)] = 1
    return cv2.bitwise_or(white, yellow), white, yellow
```



Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The main challenge for the perspective transform for this problem is finding the four source points. Looking at the road images above, the the lane lines with the bottom of the image approximately resembles a trapezoid. Ideally, two of the vertex points fall just left of the left lane and the other two vertex points fall just right of the right lane. Finding the source points required n steps.

The first step is to convert the image to binary using the algorithm explained in the previous question.

The second step is to perform a default trapezoidal crop of the image to simplify calculations. This trapezoid is represented by four vertices. Let *xsize* and *ysize* be the number of pixels in the x and y directions, respectively. Then the lower left vertex is (1 / 32 * *xsize*, *ysize*), the lower right vertex is (31 / 32 *xsize*, *ysize*), the upper left vertex is (14.5 / 32 * *xsize*, *ysize* / 1.65), and the upper right vertex is (17.5 / 32 * *xsize*, *ysize* / 1.65). Note that these values were found through trial and error.

The third step is to identify the lane lines to find the source points. This was done using the same technique as in the first lane detection project, using OpenCV's HoughLinesP() method. Outlier

lines are rejected, the average negative and positive slope lines are calculated, and a weighted mean of the x values for each the left and right lanes are calculated. The two lane lines are then found with point-slope form calculations. After identifying the two lane lines, two of the vertex points just left of the left lane and two vertex points just right of the right lane were chosen as the source points.

The last step is to pass the source and destination points in to OpenCV's getPerspectiveTransform() and pass the perspective transform along with the original image into OpenCV's warpPerspective() function. Note that the destination points are constantly the four corners of the original image.

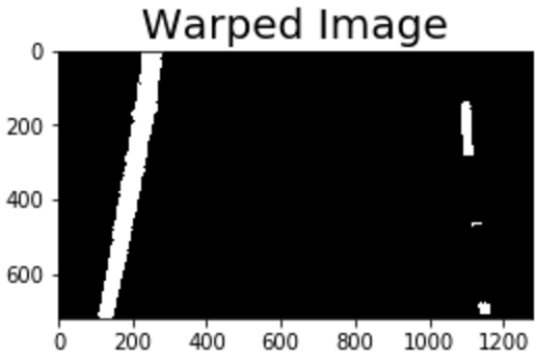
Here is an example of a perspective transform result along with my code (located in the Perspective Transform section of my Jupyter Notebook).

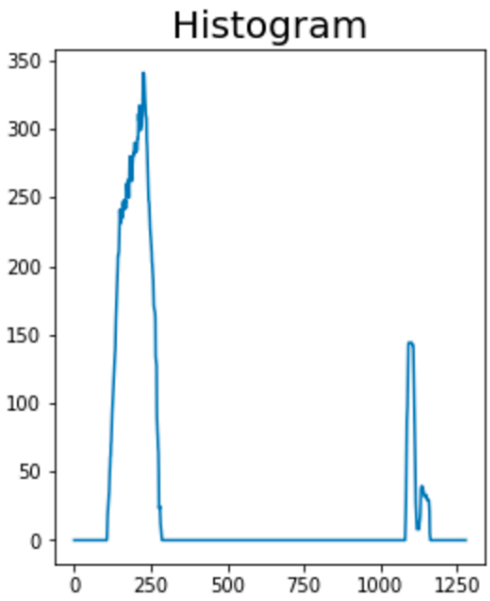

```
def birds_eye(img, src='default', dst='default'):
    offset = 0 # offset for dst points
    # Grab the image shape
    img_size = (img.shape[1], img.shape[0])
    # Grab the outer four detected corners
    if src == 'default':
        src = np.float32([corners[0], corners[nx-1], corners[-1], corners[-nx]])
    # For destination points, choose four corners of original image
    if dst == 'default':
        # top left, top right, bottom right, bottom left
        dst = np.float32([[offset, offset], [img_size[0] - offset, offset],
                                    [img_size[0] - offset, img_size[1] - offset],
                                    [offset, img_size[1] - offset]])
    # Given src and dst points, calculate the perspective transform matrix
    M = cv2.getPerspectiveTransform(src, dst)
    Minv = cv2.getPerspectiveTransform(dst, src)
    # Warp the image using OpenCV warpPerspective()
    warped = cv2.warpPerspective(img, M, img_size)

    # Return the resulting image
    return warped, Minv
```

```python
def calc_src(curr_img, ksize=3, rho=1, theta=np.pi/180, threshold=50, min_line_len=20, max_line_gap=40):
    # convert to binary image with detected lanes
    binary, _, _ = lane_binary(curr_img)
    # crop into trapezoid
    tmp = default_crop(binary)
    # find hough lines
    lines, processed_image = hough_lines(tmp, rho, theta, threshold, min_line_len, max_line_gap)
    # define variables to find point-slope form lines
    total_pos_x, total_pos_y = 0.0, 0.0
    pos_x_count, pos_y_count = 0, 0
    total_neg_x, total_neg_y = 0.0, 0.0
    neg_x_count, neg_y_count = 0, 0
    total_neg_slope, total_pos_slope = 0.0, 0.0
    neg_slope_count, pos_slope_count = 0, 0
    min_y, max_y = 600, -1
    test_lines = []
    for line in lines:
        for x1, y1, x2, y2 in line:
            if x1 - x2 == 0:
                continue
            m = float((y1 - y2)) / (x1 - x2)
            # filter out extreme slopes
            if abs(m) < 0.33 or abs(m) > 1:
                continue
            min_y = min(min_y, y1, y2)
            max_y = max(max_y, y1, y2)
            test_lines.append(line)
            if m < 0:
                total_neg_x += x1 + x2
                total_neg_y += y1 + y2
                neg_x_count += 2
                neg_y_count += 2
                total_neg_slope += m
                neg_slope_count += 1
            else:
                total_pos_x += x1 + x2
                total_pos_y += y1 + y2
                pos_x_count += 2
                pos_y_count += 2
                total_pos_slope += m
                pos_slope_count += 1
    # if one of the lane is not detected, return None
    if pos_slope_count == 0 or neg_slope_count == 0:
        return None
    # find average x, y points and average slopes
    if neg_x_count != 0 and neg_y_count != 0:
        avg_neg_x, avg_neg_y = total_neg_x / neg_x_count, total_neg_y / neg_y_count
    if pos_x_count != 0 and pos_y_count != 0:
        avg_pos_x, avg_pos_y = total_pos_x / pos_x_count, total_pos_y / pos_y_count
    if neg_slope_count != 0 and pos_slope_count != 0:
        avg_neg_slope, avg_pos_slope = total_neg_slope / neg_slope_count, total_pos_slope / pos_slope_count

    pos_slope_line = calc_line((avg_pos_x, avg_pos_y), avg_pos_slope, min_y, max_y)
    neg_slope_line = calc_line((avg_neg_x, avg_neg_y), avg_neg_slope, min_y, max_y)

    new_lines = np.array([np.array([pos_slope_line, neg_slope_line])])

    offset = 110
    # bottom left
    v1 = (new_lines[0][1][2] - offset, max_y)
    min_x = new_lines[0][1][2] - offset
    # bottom right
    v2 = (new_lines[0][0][2] + offset, max_y)
    max_x = new_lines[0][0][2] + offset
    # top right
    v3 = ((new_lines[0][0][2] - new_lines[0][0][0]) / 8 + new_lines[0][0][0] + offset / 2.5, 1 / 8 * (max_y - min_y) + min_y)
    # top left
    v4 = (new_lines[0][1][0] - (new_lines[0][1][0] - new_lines[0][1][2]) / 8 - offset / 2.5, 1 / 8 * (max_y - min_y) + min_y)
    vertices1 = np.array([[v1, v2, v3, v4]])
    res = np.float32([v4, v3, v2, v1])
    return res
```

Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial.

The warped image in the previous section is first converted to a binary image that emphasizes the lane lines using the algorithm described a couple of questions ago. Then, a histogram along the x axis is calculated such that the values of the histogram correspond to the number of pixels at that x value. For example, this is a warped binary image and its corresponding histogram:





The lane-line pixels are found using a sliding window. With the histogram represented as a numpy array, the two modes can be easily found by starting at the midpoint and iterating outward. These two modes are used as the starting points for the sliding windows. Just as in the class example, each lane is found with 9 windows each, the windows have a width of 100, and a minimum of 50 pixels inside the window is required to recenter the window.

Using this technique, the left and right line-lane pixels are effectively identified by keeping track of nonzero values inside the windows. For a first attempt, this was done blindly for each frame. The nonzero pixels values are then used to calculate the second degree polynomial coefficients using numpy's polyfit() function, and these coefficients are used to fit the polynomial to all y values.

Here is my code (located in the find_curvature() function of the Determine Lane Curvature section of my Jupyter Notebook), where left_lane_inds and right_lane_inds are the indices for the left and right lane, respectively:

```
def find_curvature(img, binary_warped, Minv):
    # binary_warped = remove_noise(img, is_binary=False)
    histogram = np.sum(binary_warped[binary_warped.shape[0]/2:,:], axis=0)
    out_img = np.dstack((binary_warped, binary_warped, binary_warped))*255
    midpoint = np.int(histogram.shape[0]/2)

    # left mode
    leftx_base = np.argmax(histogram[:midpoint])
    # right mode
    rightx_base = np.argmax(histogram[midpoint:]) + midpoint
    # number of windows
    nwindows = 9
    window_height = np.int(binary_warped.shape[0]/nwindows)

    # returns indices of elements that are non-zero
    nonzero = binary_warped.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])

    # current left mode
    leftx_current = leftx_base
    # current right mode
    rightx_current = rightx_base

    # width of window is +/- margin
    margin = 100
    # minimum number of pixels found to recenter window
    minpix = 50

    # left lane pixel indices
    left_lane_inds = []
    # right lane pixel indices
    right_lane_inds = []

    for window in range(nwindows):
        # bottom of window
        win_y_low = binary_warped.shape[0] - (window+1)*window_height
        # top of window
        win_y_high = binary_warped.shape[0] - window*window_height
        # left of left window
        win_xleft_low = leftx_current - margin
        # right of left window
        win_xleft_high = leftx_current + margin
        # left of right window
        win_xright_low = rightx_current - margin
        # right of right window
        win_xright_high = rightx_current + margin

        # output image, two corners, color, thickness
        cv2.rectangle(out_img,(win_xleft_low,win_y_low),(win_xleft_high,win_y_high),(0,255,0), 2)
        cv2.rectangle(out_img,(win_xright_low,win_y_low),(win_xright_high,win_y_high),(0,255,0), 2)
```

```
    # identify nonzero pixels in x and y within the window
    good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) & (nonzerox >= win_xleft_low) & (nonzerox < win_xleft_high)).nonzero()[0]
    good_right_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) & (nonzerox >= win_xright_low) & (nonzerox < win_xright_high)).nonzero()[0]

    # record nonzero pixels in x and y for both left and right lanes
    left_lane_inds.append(good_left_inds)
    right_lane_inds.append(good_right_inds)

    # if found > minpix pixels, recenter next window on their mean position
    if len(good_left_inds) > minpix:
        leftx_current = np.int(np.mean(nonzerox[good_left_inds]))
    if len(good_right_inds) > minpix:
        rightx_current = np.int(np.mean(nonzerox[good_right_inds]))
```

```
# convert left_lane_inds to 1-dimensional numpy array
# convert right_lane_inds to 1-dimensional numpy array
left_lane_inds = np.concatenate(left_lane_inds)
right_lane_inds = np.concatenate(right_lane_inds)

# nonzero x indices of left lane
leftx = nonzerox[left_lane_inds]
# nonzero y indices of left lane
lefty = nonzeroy[left_lane_inds]
# nonzero x indices of right lane
rightx = nonzerox[right_lane_inds]
# nonzero y indices of right lane
righty = nonzeroy[right_lane_inds]

# if a lane is missing, return None
if len(lefty) == 0 or len(leftx) == 0 or len(righty) == 0 or len(rightx) == 0:
    return None, None, 0, 0, 0

# get second degree polynomial coefficients for points on left lane
left_fit = np.polyfit(lefty, leftx, 2)
# get second degree polynomial coefficients for points on right lane
right_fit = np.polyfit(righty, rightx, 2)

# generate evenly spaced numbers from 0 to largest y value
ploty = np.linspace(0, binary_warped.shape[0]-1, binary_warped.shape[0])
# fit second degree polynomial for left lane
left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
# fit second degree polynomial for right lane
right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]
```

Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to the center.

Calculating the radius of curvature is fairly straightforward after determining the polynomial coefficients. For each lane, the radius of curvature is calculated with the following formula:

$$R_{curve} = \frac{(1+(2Ay+B)^2)^{3/2}}{|2A|}$$

where A and B are the coefficients for the second and first degree, respectively.

Finding the center of the vehicle is done by first drawing the lane on a perspective transformed blank image, warping it with the inverse perspective transform, and finding the lower left and right corners of this new warp image. Assuming the camera is centered, the midpoint between the lower left and right corners denotes the vehicle's position. This midpoint is then compared to the middle x value of the image (newwarp_gray.shape[1] / 2) to find the offset.

Here is the relevant portion of the code that calculates both the curvatures and vehicle's position with respect to the center (located in the find_curvature() function of the Determine Lane Curvature section of my Jupyter Notebook):

```
y_eval = np.max(ploty)
# calculate left curvature
left_curverad = ((1 + (2*left_fit[0]*y_eval + left_fit[1])**2)**1.5) / np.absolute(2*left_fit[0])
# calculate right curvature
right_curverad = ((1 + (2*right_fit[0]*y_eval + right_fit[1])**2)**1.5) / np.absolute(2*right_fit[0])

# Create an image to draw the lines on
warp_zero = np.zeros_like(binary_warped).astype(np.uint8)
color_warp = np.dstack((warp_zero, warp_zero, warp_zero))

# Recast the x and y points into usable format for cv2.fillPoly()
pts_left = np.array([np.transpose(np.vstack([left_fitx, ploty]))])
pts_right = np.array([np.flipud(np.transpose(np.vstack([right_fitx, ploty])))])
pts = np.hstack((pts_left, pts_right))

# Draw the lane onto the warped blank image
cv2.fillPoly(color_warp, np.int_([pts]), (0,255, 0))

# Warp the blank back to original image space using inverse perspective matrix (Minv)
newwarp = cv2.warpPerspective(color_warp, Minv, (warp_zero.shape[1], warp_zero.shape[0]))
newwarp_gray = cv2.cvtColor(newwarp, cv2.COLOR_RGB2GRAY)

# find lower left and right corners of newwarp region
# used to calculate vehicles's offset from center
leftmost_x = newwarp_gray.shape[1]
rightmost_x = -1
flag = False
for i in range(newwarp_gray.shape[0] - 1, -1, -1):
    if flag:
        break
    for j in range(newwarp_gray.shape[1] - 1, -1, -1):
        if newwarp_gray[i][j] != 150:
            continue
        flag = True
        leftmost_x = min(leftmost_x, j)
        rightmost_x = max(rightmost_x, j)

# convert pixels to meters
meters_per_pix = 3.7 / 700
# absolute center - car's center
center = meters_per_pix * (img.shape[1] / 2 - (rightmost_x + leftmost_x) / 2)
# left curve radius
left_curverad = meters_per_pix * left_curverad
# right curve radius
right_curverad = meters_per_pix * right_curverad
# Combine the result with the original image
result = cv2.addWeighted(img, 1, newwarp, 0.3, 0)
return result, newwarp, left_curverad, right_curverad, center
```

Note that there are 3.7 pixels / 700 meters, according to Udacity forums. So, the center offset, left_curverad, and right_curverad are each multiplied by 3.7 / 700 to covert to meters.

Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

Original Image

Image with Lane Detection

Discussion

Briefly discuss any problem / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

Most of this project was explained very well in the Udacity lessons. Of everything, I found thresholding to be the biggest challenge because there are so many different options that require trial and error.

At first, I thresholded with Sobel and RGB (red/green/blue), but I had trouble getting the correct threshold values for clean lane detection. Next, I tried a combination of RGB and HLS (hue/lightness/saturation) color space, but this did not work well either. After researching HLS, I learned that thresholding with RGB isn't as easy as simply thresholding with H since H encompasses the entire color wheel. So, in the end, I used only HLS thresholding. I looked online for a color wheel for H and found that yellow and orange can easily be isolated within range [11, 33]; this easily detected the yellow lanes. Next, I tried detecting white lanes and, after some research, found that only L is necessary for this; L in range [198, 255] is sufficient for white lanes.

While this appeared to work, the thresholding did not effectively filter out bridges and shadows. Bridges are also slightly yellow which threw things off. After some analysis, it was clear that the yellow lanes are more saturated than the bridge color, so thresholding S in range [86, 255] resolved the bridge issue. For yellow lanes, thresholding L in range [20, 255] effectively filtered out shadows.

The next most challenging part was dealing with situations when the lane detection algorithm either did not detect enough source points for the perspective transform or did not detect lane lines for fitting the polynomial. When the source points were not detected, I reused the previous source points which worked well. When the polynomial could not be fitted, I reused the previous lane detection area which worked well also. Because my algorithm currently reuses previously calculated values in such situations, my pipeline is likely to fail when either the very first frame has lane lines that are difficult to detect or when consecutive frames have lane lines that are difficult to detect.

There are a few ways that can make this algorithm more robust. Currently, the algorithm performs a blind search for each video frame which not only is computationally expensive, but may be less inaccurate. This can be improved upon by caching the polynomial coefficients, or blindly searching in areas that have not already been searched. Also, I suspect that thresholding with Sobel in addition to HLS will lead to more robust lane detection; this will be experimented with in future iterations.