Pipeline

March 7, 2017

0.1 Advanced Lane Finding Project

In [1]:
```
import cv2
import os
import numpy as np
import matplotlib.pyplot as plt
from scipy.misc import imread as imread
from scipy.misc import imsave as imsave
from scipy.misc import imresize as imresize
from IPython.display import HTML
from moviepy.editor import VideoFileClip
from moviepy.editor import ImageSequenceClip
import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
%matplotlib inline
```

0.2 Camera Calibration

In [5]:
```
# Load chessboard images for camera callibration

folder = 'camera_cal/'
image_files = os.listdir(folder)
chessboard_images = []

for i in range(len(image_files)):
    file_name  = os.path.join(folder, ('calibration' + str(i+1) + '.jpg'))
    image_file = imread(file_name, False, 'RGB')
    image_file = imresize(image_file, (720, 1280), interp='bilinear')
    chessboard_images.append(image_file)
```

In [6]:
```
# Compute the camera calibration matrix and distortion coefficients
# given a set of chessboard images

def compute_cal_dist(img):
    # Prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
    objp = np.zeros((6*8,3), np.float32)
    objp[:,:2] = np.mgrid[0:8, 0:6].T.reshape(-1,2)
```

```
            # Convert to grayscale
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

            # Find the chessboard corners
            ret, corners = cv2.findChessboardCorners(gray, (8,6), None)

            objpoints = objp
            imgpoints = corners

            return ret, objpoints, imgpoints
```

In [7]:
```
# Arrays to store object points and image points from all the images.
objpoints = [] # 3d points in real world space
imgpoints = [] # 2d points in image plane.

for i in range(20):
    img = chessboard_images[i]
    ret, objp, imp = compute_cal_dist(img)

    # If found, add object points, image points
    if ret == True:
        objpoints.append(objp)
        imgpoints.append(imp)
```

In [9]:
```
# Do camera calibration given object points and image points
img = cv2.imread('test_image.jpg')
img_size = (img.shape[1], img.shape[0])

# Do camera calibration given object points and image points
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, im
```

In [10]:
```
def calibrate_image(image, mtx, dist):
    '''
    Applies distortion correction
    '''

    # Apply distortion correction to raw images.
    undst = cv2.undistort(image, mtx, dist, None, mtx)

    return undst
```

0.3 Pipeline Functions

In [11]:
```
def mag_thresh(img, sobel_kernel=3, mag_thresh=(0, 255)):
    '''
    To create an edge image
    '''
    # Convert to grayscale
```

```
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    # Take both Sobel x and y gradients
    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sobel_kernel)
    sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=sobel_kernel)

    # Calculate the gradient magnitude
    gradmag = np.sqrt(sobelx**2 + sobely**2)

    # Rescale to 8 bit
    scale_factor = np.max(gradmag)/255
    gradmag = (gradmag/scale_factor).astype(np.uint8)

    # Create a binary image of ones where threshold is met, zeros otherwis
    binary_output = np.zeros_like(gradmag)
    binary_output[(gradmag >= mag_thresh[0]) & (gradmag <= mag_thresh[1])]

    # Return the binary image
    return binary_output
```

In [12]:
```
def grad_threshold(img, sobel_kernel=3, thresh=(0, np.pi/2)):
    '''
    To create a gradient image
    '''
    # Grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    # Calculate the x and y gradients
    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sobel_kernel)
    sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=sobel_kernel)

    # Take the absolute value of the gradient direction,
    # apply a threshold, and create a binary image result
    absgraddir = np.arctan2(np.absolute(sobely), np.absolute(sobelx))
    binary_output =  np.zeros_like(absgraddir)
    binary_output[(absgraddir >= thresh[0]) & (absgraddir <= thresh[1])] =

    # Return the binary image
    return binary_output
```

In [13]:
```
def color(img, thresh=(0, 255)):
    '''
    To create a color map image
    '''
    # Convert to HLS color space
    hls = cv2.cvtColor(img, cv2.COLOR_RGB2HLS)

    # Select the S-channel
```

```
    S = hls[:,:,2]

    # Take the HLS color space, apply a threshold, and create a binary ima
    binary = np.zeros_like(S)
    binary[(S > thresh[0]) & (S <= thresh[1])] = 1

    return binary
```

In [14]:
```
def region_of_interest(img, vertices):
    '''
    Applies an image mask.
    Only keeps the region of the image defined by the polygon
    formed from `vertices`. The rest of the image is set to black.
    '''
    # Defining a blank mask to start with
    mask = np.zeros_like(img)

    # Defining a 3 channel or 1 channel color to fill the mask with depend
    if len(img.shape) > 2:
        channel_count = img.shape[2]  # i.e. 3 or 4 depending on your imag
        ignore_mask_color = (255,) * channel_count
    else:
        ignore_mask_color = 255

    # Filling pixels inside the polygon defined by "vertices" with the fil
    cv2.fillPoly(mask, vertices, ignore_mask_color)

    # Returning the image only where mask pixels are nonzero
    masked_image = cv2.bitwise_and(img, mask)

    return masked_image
```

In [15]:
```
def corners_unwarp(img, vx1, vx2, vy1, vy2):
    '''
    Prespective Transform
    '''
    # Input should be the calibrated Image
    undist = img

    # Grab the image shape
    img_size = (img.shape[1], img.shape[0])

    # For source points:
    src = np.float32([vx1, vx2, vy1, vy2])

    # For destination points:
    dst = np.float32([vx1, [vx1[0], 0], [vy2[0], 0], vy2])
```

```
    # Given src and dst points, calculate the perspective transform matrix
    M = cv2.getPerspectiveTransform(src, dst)

    # Compute the inverse perspective transform:
    Minv = cv2.getPerspectiveTransform(dst, src)

    # Warp the image using OpenCV warpPerspective()
    warped = cv2.warpPerspective(undist, M, img_size)

    # Return the resulting image and matrix
    return warped, M, Minv
```

In [16]:
```
def binary_map_norm(img):
    '''
    To normalize the binary image map
    '''
    # Find the min value in the array
    img = np.array(img)
    min_value = np.min(img)

    # Loop through each element of the image to replace non-zero elements
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            if img[i][j] >= min_value and img[i][j]!= 0:
                img[i][j] = 255.0

    img = img/255.0

    return img
```

In [17]:
```
def curvature(top_down, left_lane_inds, right_lane_inds, left_fit, right_f
    '''
    Find the lane curvature
    '''
    # Identify the x and y positions of all nonzero pixels in the image
    nonzero  = top_down.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])

    lefty  = nonzeroy[left_lane_inds]
    righty = nonzeroy[right_lane_inds]

    y_eval1 = np.max(lefty)
    y_eval2 = np.max(righty)

    left_curverad = ((1 + (2*left_fit[0]*y_eval1 + left_fit[1])**2)**1.5)
    right_curverad = ((1 + (2*right_fit[0]*y_eval2  + right_fit[1])**2)**1

    return left_curverad, right_curverad
```

```
In [18]: def first_lane(image, vertices):
             '''
             To find the lane in the first frame of video sequence.
             '''
             # Apply binary map to the next images.
             b_map = mag_thresh(image, sobel_kernel=3, mag_thresh=(40, 150))

             # Gradient Orientation
             grad  = grad_threshold(image, sobel_kernel=15, thresh=(0.7, 1.3))

             # Color
             color_map = color(image, thresh=(90, 255))

             # Combine
             combined = np.zeros_like(grad)
             combined[((b_map == 1)) | ((color_map == 1) & (grad == 1))] = 1

             masked_edge_image = region_of_interest(combined, vertices)
             masked_image = region_of_interest(image, vertices)

             top_down, M, Minv = corners_unwarp(masked_edge_image, vertices[0,0,:],
             top_down = binary_map_norm(top_down)

             # Compute the histogram
             histogram = np.sum(top_down[top_down.shape[0]/2:,:], axis=0)

             # Find the peak of the left and right halves of the histogram
             # These will be the starting point for the left and right lines
             midpoint    = np.int(histogram.shape[0]/2)
             leftx_base  = np.argmax(histogram[:midpoint])
             rightx_base = np.argmax(histogram[midpoint:]) + midpoint

             # Choose the number of sliding windows
             nwindows = 9

             # Set height of windows
             window_height = np.int(top_down.shape[0]/nwindows)

             # Identify the x and y positions of all nonzero pixels in the image
             nonzero  = top_down.nonzero()
             nonzeroy = np.array(nonzero[0])
             nonzerox = np.array(nonzero[1])

             # Current positions to be updated for each window
             leftx_current  = leftx_base
             rightx_current = rightx_base

             # Set the width of the windows +/- margin
```

```
margin = 100

# Set minimum number of pixels found to recenter window
minpix = 50

# Create empty lists to receive left and right lane pixel indices
left_lane_inds = []
right_lane_inds = []

# Step through the windows one by one
for window in range(nwindows):
    # Identify window boundaries in x and y (and right and left)
    win_y_low = top_down.shape[0] - (window+1)*window_height
    win_y_high = top_down.shape[0] - window*window_height
    win_xleft_low = leftx_current - margin
    win_xleft_high = leftx_current + margin
    win_xright_low = rightx_current - margin
    win_xright_high = rightx_current + margin

    # Identify the nonzero pixels in x and y within the window
    good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high
    good_right_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_hig

    # Append these indices to the lists
    left_lane_inds.append(good_left_inds)
    right_lane_inds.append(good_right_inds)

    # If you found > minpix pixels, recenter next window on their mean
    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]))

# Concatenate the arrays of indices
left_lane_inds  = np.concatenate(left_lane_inds)
right_lane_inds = np.concatenate(right_lane_inds)

# Extract left and right line pixel positions
leftx  = nonzerox[left_lane_inds]
lefty  = nonzeroy[left_lane_inds]
rightx = nonzerox[right_lane_inds]
righty = nonzeroy[right_lane_inds]

# Fit a second order polynomial to each
left_fit = np.polyfit(lefty, leftx, 2)
right_fit = np.polyfit(righty, rightx, 2)

# Find the curve:
```

```
    left_curverad, right_curverad = curvature(top_down, left_lane_inds, ri

    # Generate x and y values for plotting
    ploty = np.linspace(0, top_down.shape[0]-1, top_down.shape[0] )
    left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
    right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]

    # Create an image to draw the lines on
    font       = cv2.FONT_HERSHEY_SIMPLEX
    warp_zero  = np.zeros_like(masked_edge_image).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, pl
    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))
    cv2.putText(image,'Radius of Curvature: '  + str(left_curverad) + 'm',

    # Warp the blank back to original image space using inverse perspectiv
    newwarp = cv2.warpPerspective(color_warp, Minv, (image.shape[1], image

    # Combine the result with the original image
    result = cv2.addWeighted(image, 1, newwarp, 0.3, 0)

    return result, left_fit, right_fit
```

In [19]:
```
def second_to_end_lane(image, vertices, left_fit, right_fit):
    '''
    To find the lane from second frame to end
    '''

    # Apply binary map to the next images.
    b_map = mag_thresh(image, sobel_kernel=3, mag_thresh=(40, 150))

    # Gradient Orientation
    grad  = grad_threshold(image, sobel_kernel=15, thresh=(0.7, 1.3))

    # Color
    color_map = color(image, thresh=(90, 255))

    # Combine
    combined = np.zeros_like(grad)
    combined[((b_map == 1)) | ((color_map == 1) & (grad == 1))] = 1

    masked_edge_image = region_of_interest(combined, vertices)
```

```
masked_image = region_of_interest(image, vertices)

top_down, M, Minv = corners_unwarp(masked_edge_image, vertices[0,0,:],
top_down = binary_map_norm(top_down)

nonzero  = top_down.nonzero()
nonzeroy = np.array(nonzero[0])
nonzerox = np.array(nonzero[1])
margin   = 100
left_lane_inds  = ((nonzerox > (left_fit[0]*(nonzeroy**2) + left_fit[1
right_lane_inds = ((nonzerox > (right_fit[0]*(nonzeroy**2) + right_fit

# Again, extract left and right line pixel positions
leftx  = nonzerox[left_lane_inds]
lefty  = nonzeroy[left_lane_inds]
rightx = nonzerox[right_lane_inds]
righty = nonzeroy[right_lane_inds]

# Fit a second order polynomial to each
left_fit  = np.polyfit(lefty, leftx, 2)
right_fit = np.polyfit(righty, rightx, 2)

# Generate x and y values for plotting
ploty = np.linspace(0, top_down.shape[0]-1, top_down.shape[0] )
left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]

# Generate a polygon to illustrate the search window area
# And recast the x and y points into usable format for cv2.fillPoly()
left_line_window1  = np.array([np.transpose(np.vstack([left_fitx-margi
left_line_window2  = np.array([np.flipud(np.transpose(np.vstack([left_
left_line_pts      = np.hstack((left_line_window1, left_line_window2))
right_line_window1 = np.array([np.transpose(np.vstack([right_fitx-marg
right_line_window2 = np.array([np.flipud(np.transpose(np.vstack([right
right_line_pts     = np.hstack((right_line_window1, right_line_window2

# Find the curve:
left_curverad, right_curverad = curvature(top_down, left_lane_inds, ri

# Create an image to draw the lines on
font       = cv2.FONT_HERSHEY_SIMPLEX
warp_zero  = np.zeros_like(masked_edge_image).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, pl
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))
            cv2.putText(image,'Radius of Curvature: '  + str(left_curverad) + 'm',

            # Warp the blank back to original image space using inverse perspectiv
            newwarp = cv2.warpPerspective(color_warp, Minv, (image.shape[1], image

            # Combine the result with the original image
            result = cv2.addWeighted(image, 1, newwarp, 0.3, 0)

            return result, left_fit, right_fit
```

0.4 Main Pipeline

0.4.1 Load Video and Extract frames

```
In [20]: # Load Video
         clip = VideoFileClip("project_video.mp4")

         # Get the frames from the video
         frames = []
         for x in clip.iter_frames():
             frames.append(x)

         # Convert the sequence to numpy array
         frames = np.array(frames)
         print('Frame Sequence shape: ', frames.shape)
```

```
Frame Sequence shape:  (1260, 720, 1280, 3)
```

0.4.2 Lane Finding Pipeline

```
In [21]: # Define Vertices:
         vertices = np.array([[(240,720),(580, 440), (700, 440), (1200,720)]], dtyp

         # To save
         lane_map = []

         print('Starting: ', end=" ")

         # Get first frame
         frame = frames[0, :, :, :]

         # Correct for distortion
         undist_frame = calibrate_image(frame, mtx, dist)

         # Find the lane
```

```
        result, left_fit, right_fit = first_lane(undist_frame, vertices)

        # Append the final result
        lane_map.append(result)

        print ('>', end= " ")

        # From Second frame to the end
        for i in range(1, frames.shape[0]):
            # Get frame
            frame = frames[i, :, :, :]

            # Correct for distortion
            undist_frame = calibrate_image(frame, mtx, dist)

            # Find the lane
            result, left_fit, right_fit = second_to_end_lane(undist_frame, vertice

            # Append the final result
            lane_map.append(result)

            # Progress Bar, every 100 frames.
            if i%100 == 0:
                print ('>', end= " ")

        print ('end', end=" ")
```

```
Starting:  > > > > > > > > > > > > > end
```

0.4.3 Save the output to output_images directory

```
In [22]: # Save the output image
         print('Saving: >', end=" ")

         for j in range(len(lane_map)):
             # Get the output image
             image = lane_map[j]

             # Filename to save
             name  = 'output_images/'+(str(j)+'.jpg')

             # Save the image to directory
             imsave(name, image)

             # Increment counter
             j += 1

             # Progress Bar, every 100 frames
```

```
            if j%100 == 0:
                print ('>', end= " ")

        print ('end', end=" ")
```

```
Saving: > > > > > > > > > > > > > end
```

0.4.4 Convert Output Frames to Video

```
In [12]: # Define the codec and create VideoWriter object

         import skvideo.io

         writer = skvideo.io.FFmpegWriter("output.mp4", outputdict={'-vcodec': 'lib

         for i in range(1260):
             filename = 'output_images/' + str(i) + '.jpg'
             read = imread(filename, False, 'RGB')
             read = np.array(read)

             # Write to video
             writer.writeFrame(read)

         writer.close()
```

```
In [3]: HTML("""
        <video width="320" height="200" controls>
          <source src="{0}">
        </video>
        """.format("output.mp4"))
```

```
Out[3]: <IPython.core.display.HTML object>
```