Advanced Lane Finding Project

Paul Comitz
9/5/2017

The goals / steps of this project are the following:

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birds-eye view").
- Detect lane pixels and fit to find the lane boundary.
- Determine the curvature of the lane and vehicle position with respect to center.
- Warp the detected lane boundaries back onto the original image.
- Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

These steps are discussed in the notebook sections below. The cells that immediately follow this introduction describes the functions that were developed to process the individual images in this project. The images are extractedfrom the supplied project video, using code such as :

```
"""
proj4_output = 'proj4.mp4'
clip1 = VideoFileClip('project_video.mp4')
proj4_clip = clip1.fl_image(process_image)
proj4_clip.write_videofile(proj4_output, audio=False)
# %time for jupyter
"""
The code above prodcues 1261 individual images. The code above is not runs as part of this notbo
ok due to the lengthy output. The video that is produced > proj4.mp4 is available in the submitt
ed github repository.
```

Perspective Transform Function

A perspective transform maps the points in a given image to different, desired, image points with a new perspective. The perspective transform used in this project is a bird's-eye view transform that provides a view of a lane from above. This is used for calculating the lane curvature later in the project. The function below provides a way to transfrom an image from the source image to the destination image or perfrom the transfrom from the destination image back to the source image.

In [3]:

```
#### Perspective Transfrom
#From project examples
# Returns warped or unwarped image
#based on setting of inv flag
####
def warper(img,inv = False):
    # Perspective transform
    # choose the source and destination coordinates
    # used test2.jpg , use the deer crossing
    # sign as a reference
    img_size = (img.shape[1], img.shape[0])
    p1x = (img_size[0]/2 -55)
    p1y = img_size[1]/2 + 100
    p2x = (img_size[0]/6) -10
    p2y = img_size[1]
    p3x = (img_size[0] * 5 /6) + 60
    p3y = img_size[1]
    p4x = (img_size[0]/2 + 55)
    p4y = img_size[1]/2 + 100

    d1x = (img_size[0]/4)
    d1y = 0
    d2x = (img_size[0]/4)
    d2y = img_size[1]
    d3x = (img_size[0]*3 /4)
    d3y = img_size[1]
    d4x = (img_size[0]*3 /4)
    d4y = 0

    src = np.float32(
    [[p1x,p1y],
     [p2x,p2y],
     [p3x, p3y],
     [p4x,p4y]])

    dst = np.float32(
    [[d1x,d1y],
     [d2x,d2y],
     [d3x,d3y],
     [d4x,d4y]])

    # Compute and apply perpective transform
    if (inv == False):
        M = cv2.getPerspectiveTransform(src, dst)
    elif (inv == True):
        M= cv2.getPerspectiveTransform(dst,src)

    # keep same size as input image
    warped = cv2.warpPerspective(img, M, img_size, flags=cv2.INTER_NEAREST)
    return warped
```

Sobel Transform

Canny edge detection finds *all* lines in an image The sobel transform computes gradients that define steep edges. The steep edges are likely to be lanes.

- take derivative in x or y direction. Assume lane lines near vertical, take x gradient.
- kernel size default is 3 x 3, larger kernel is a smoother gradient
- x direction emphasizes images closer to vertical
- y direction closer to horizontal

In [8]:

```
#### Sobel Threshold
# Define a function that applies Sobel x or y,
# Takes a color image and converts to gray scale
# returns a gray scale image
# then takes an absolute value and applies a threshold.
# Note: calling function with orient='x', thresh_min=5, thresh_max=100
# is a typical usage
####
def abs_sobel_thresh_phc(img, orient='x', thresh_min=0, thresh_max=255):

    # Apply the following steps to img
    # 1) Convert to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    # 2) Take the absolute value of the derivative in x or y
    #    given orient = 'x' or 'y'
    if(orient == 'x'):
        abs_sobel= np.absolute(cv2.Sobel(gray, cv2.CV_64F, 1, 0))
    else:
        abs_sobel = np.absolute(cv2.Sobel(gray, cv2.CV_64F, 0, 1))

    # 4) Scale to 8-bit (0 - 255) then convert to type = np.uint8
    scaled_sobel = np.uint8(255*abs_sobel/np.max(abs_sobel))

    # 5) Create a mask of 1's where the scaled gradient magnitude
            # is > thresh_min and < thresh_max
    sxbinary = np.zeros_like(scaled_sobel)
    sxbinary[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1

    # 6) Return this mask as your binary_output image
    # binary_output = np.copy(img) # Remove this line
    binary_output = sxbinary
    return binary_output
```

Magnitude of the Gradient Function

The magnitide of the gradient is given by:

- square root (square(Sx) + square(Sy))

In [18]:

```
#### Magnitude Gradient
# Define a function to return the magnitude of the gradient
# for a given sobel kernel size and threshold values
####
def mag_thresh(img, sobel_kernel=3, mag_thresh=(0, 255)):
    # 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 otherwise
    binary_output = np.zeros_like(gradmag)
    binary_output[(gradmag >= mag_thresh[0]) & (gradmag <= mag_thresh[1])] = 1

    # Return the binary image
    return binary_output
```

Direction of the Gradient

This function computes the direction of the gradient by taking the arctan of the absolute value of the sobel x and sobel y gradients.

In [20]:
```
#### Direction of Gradient
# Define a function that applies Sobel x and y,
# then computes the direction of the gradient
# and applies a threshold.
####
def dir_threshold(img, sobel_kernel=3, thresh=(0, np.pi/2)):

    # Apply the following steps to img
    # 1) Convert to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    # 2) Take the gradient in x and y separately
    #sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sobel_kernel)
    #sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=sobel_kernel)
    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sobel_kernel)
    sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=sobel_kernel)
    # 3) Take the absolute value of the x and y gradients
    abs_sobelx = np.absolute(sobelx)
    abs_sobely = np.absolute(sobely)
    # 4) Use np.arctan2(abs_sobely, abs_sobelx) to calculate the direction of the gradient
    #direction = np.arctan2(abs_sobely,abs_sobely)
    direction = np.arctan2(np.absolute(sobely), np.absolute(sobelx))
    # 5) Create a binary mask where direction thresholds are met
    # Return an array of zeros with the same shape and type as a given array.
    binary_output =  np.zeros_like(direction)
    binary_output[(direction >= thresh[0]) & (direction <= thresh[1])] = 1
    # 6) Return this mask as your binary_output image
    # binary_output = np.copy(img) # Remove this line
    return binary_output
```

Combine Thresholds

This function takes a color image and divided the image into three channels: H, L, and S using the OpenCV function: `hls = cv2.cvtColor(img,cv2.COLOR_RGB2HLS).astype(np.float)`. The S chanel is thresholded and combined with a scaled Sobel x direction gradient. The gray scaled thresholded sobel and the thresholded S channel are combined and returned.

In [27]:

```
#### Combine Sobel and color threshold
# img is the undistorted color image
# returns an array of two images
# first image is uint combined x gradient and thresholded S (saturation)
#
#  second image is color binary float 64
#   green channel is gradient
#   blue channel is thresholded s channel
#
#  NOTE: This version was changed return a single image - sobelx and s combined
####
def pipeline_sobel_hls (img, s_thresh_min=170, s_thresh_max = 255,
                        thresh_min=20,
                        thresh_max=100,
                        sobel_kernel=3):
    # from in-class class examples
    img = np.copy(img)

   # 1) covert to HLS and separate S channel
    hls = cv2.cvtColor(img,cv2.COLOR_RGB2HLS).astype(np.float)
    h_channel = hls[:,:,0]
    l_channel = hls[:,:,1]
    s_channel = hls[:,:,2]

    #2) Get the grayscale image
    gray = cv2.cvtColor(img,cv2.COLOR_RGB2GRAY)

    # 3) sobel x - take x derivative of grayscale
    # lane lines are near vertical
    # absolute x value is to accentuate lines away from horizontal
    abs_sobelx= np.absolute(cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sobel_kernel))

    # 4) scale and normalize
    scaled_sobel = np.uint8(255*abs_sobelx/np.max(abs_sobelx))

    #5) Apply threshold to x gradient
    #   color = 1 if grayscale between thresh min and thresh max
    sxbinary = np.zeros_like(scaled_sobel)
    sxbinary[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1

    #6) apply threshold to S color channel
    # shape of s_channel is (720,1280)
    s_binary = np.zeros_like(s_channel)
    s_binary[(s_channel >= s_thresh_min) & (s_channel <= s_thresh_max)] = 1

    #7)Combined binary gradient and S Channel
    combined_binary = np.zeros_like(sxbinary)
    combined_binary[(s_binary == 1) | (sxbinary == 1)] = 1


    return combined_binary
```

Sliding Window

As described in class, the sliding windows techniues is used to map out the lane lines. The high level steps for the procedure are as follows:

- histogram of thresholded binary image
- identify the peaks in the historgarm
- Start to calculate the wiondow prositions at the peaks
- Calculate the coordinates of the windows alongs the lane lines. As suggeested in class 9 windows are used. With an image that is 720 pixels in the y axis, each window is 80 pixels in the y dimesions. The x coordinates are calculated based on the histogram peaks.

In [51]:

```
#### Sliding Window
# Function for sliding window approch to mapping lanes lines
# This approach largely follows the techniques shown in class. Various
# minor modificatiosn are taken from interactions in the forum.
####
def slide_window(binary_warped,nwindows=9):

    # create an output image to draw the windows on and visualize result
    out_img = np.dstack((binary_warped, binary_warped, binary_warped))*255

    # take histogram of bottom half of image
    histogram = np.sum(binary_warped[binary_warped.shape[0]//2:,:], axis=0)

    # find the peak of the left and right halves of histogram
    # these are the starting points for the left and right lanes
    midpoint = np.int(histogram.shape[0]/2)
    leftx_base = np.argmax(histogram[:midpoint])
    rightx_base = np.argmax(histogram[midpoint:]) + midpoint

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

    # identify x and y positions of all nonzero (= white for binary image)
    # pixels in image
    nonzero = binary_warped.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 width of 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):
        win_y_low = binary_warped.shape[0] - (window+1)*window_height
        win_y_high = binary_warped.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
        print("Win: %d, %d, %d, %d, %d, %d" %(win_y_low, win_y_high, win_xleft_low,win_xlef
t_high, win_xright_low, win_xright_high))

        # windows coordinates calculated, draw windows on image, left then right
        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),(0,25
5,0), 2)

        # find the nonzero pixels in x and y in 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]

        # append the good left and right indices to the lists
        left_lane_inds= np.append(left_lane_inds,good_left_inds)
        right_lane_inds= np.append(right_lane_inds,good_right_inds)

        #Fix IndexError: arrays used as indices must be of integer (or boolean) type
        left_lane_inds = left_lane_inds.astype(int)
        right_lane_inds = right_lane_inds.astype(int)

        # if > minpix found, recenter next window on 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]))

    return out_img,histogram,left_lane_inds,right_lane_inds
```

Get Lane Pixels

The function below is a helper function to find the nonzero lane pixels in an image

In [41]:

```
#### Get Lane Pixels
# function to assist with too many indices problem
# extracts left and right line pixel positions
####
def get_lane_pixels(img):
    img_shape = img.shape
    leftx = []
    lefty = []
    rightx = []
    righty = []
    for y in range(img.shape[0]):
        for x in range(img.shape[1]):
            # print(img[y,x])
            if img[y,x] > 0:
                if (x <= img_shape[1]/2):
                    leftx.append(x)
                    lefty.append(y)
                else:
                    rightx.append(x)
                    righty.append(y)
    return leftx, lefty, rightx, righty
```

Search Region

This function provides a seacrh margin around the left and right lane pixels that are identified. The calculated areas show where the search for the lane lines was performed. This function was removed and not used. It is included here as an area for imporovemnet and future consideration.

In [57]:
```
####
# Lane lines found, search a margin around
# previous line positions
#####
def search_region(binary_warped,left_lane_inds, right_lane_inds,left_fit,right_fit):
    nonzero = binary_warped.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])
    margin = 100

    # left lane
    left_lane_inds = ((nonzerox > (left_fit[0]*(nonzeroy**2) + left_fit[1]*nonzeroy +
    left_fit[2] - margin)) & (nonzerox < (left_fit[0]*(nonzeroy**2) +
    left_fit[1]*nonzeroy + left_fit[2] + margin)))

    # right lane
    right_lane_inds = ((nonzerox > (right_fit[0]*(nonzeroy**2) + right_fit[1]*nonzeroy +
    right_fit[2] - margin)) & (nonzerox < (right_fit[0]*(nonzeroy**2) +
    right_fit[1]*nonzeroy + right_fit[2] + margin)))

    # as before extract left and right line pixel positions
    leftx,lefty,rightx,righty = get_lane_pixels(binary_warped)

    # 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, binary_warped.shape[0]-1, binary_warped.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]

    return out_img, left_lane_inds, right_lane_inds,left_fitx,right_fitx
```

Measure Curvature

Calculate radius of curvature and car offset

In [61]:
```
#### Measuring Curvature
# with help from members of forum
# Calculate radius of curvature and car offset
####

def measure_curvature(binary_warped,original_image,right_fitx,left_fitx, ploty):

    line_separation = np.mean(((right_fitx) + (left_fitx))/2)
    ym_per_pix = 30/700
    xm_per_pix = 3.7/720

    # Find offset of vehicle
    center_offset = line_separation - (binary_warped.shape[-1]//2)
    car_offset = center_offset * xm_per_pix

    #y_eval is the point where the radius of curvature is measured
    # = 719 in these images
    y_eval = np.max(ploty)

    # fit polynomials
    # Fit new polynomials to x,y
    left_fit_cr = np.polyfit(ploty*ym_per_pix, left_fitx*xm_per_pix, 2)
    right_fit_cr = np.polyfit(ploty*ym_per_pix, right_fitx*xm_per_pix, 2)

    # do radius of curvature
    left_curverad = ((1 + (2*left_fit_cr[0]*y_eval*ym_per_pix + left_fit_cr[1])**2)**1.5) /
 np.absolute(2*left_fit_cr[0])
    right_curverad = ((1 + (2*right_fit_cr[0]*y_eval*ym_per_pix +
right_fit_cr[1])**2)**1.5) / np.absolute(2*right_fit_cr[0])
    ave_curverad = (left_curverad + right_curverad)/2

    car_offset = 'Car Offset: ' + '{0:.2f}'.format(car_offset) + 'm'
    ave_curverad = 'Radius of Curvature:' + '{0:.2f}'.format(ave_curverad) + 'm'

    # unwarp the image and plot on predicted lane lines
    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 blank image , green channel
    cv2.fillPoly(color_warp, np.int_([pts]), (0,255, 0))

    # Warp the blank back to original image space using inverse perspective matrix (Minv)
    newwarp = warper(color_warp, inv = True)
    result = cv2.addWeighted(original_image, 1, newwarp, 0.3, 0)
    cv2.putText(result, car_offset , (100, 90), cv2.FONT_HERSHEY_SIMPLEX, 2, (255,255,255),
 thickness=2)
    cv2.putText(result, ave_curverad , (100, 150), cv2.FONT_HERSHEY_SIMPLEX, 2, (255,255,25
5), thickness=2)

    # save if desired
    #mpimg.imsave('output_images/finalResultImage.jpg',result)

    return result
```

Process Image Function

This function combines much of what is shown elsewhere in this writeup. It is used when making the video using the code shown on the first cell of this notebook.

In [73]:

```
def process_image(img):

    original_image = img
    print(" original np.shape(img):", np.shape(original_image))

    # retreive the camera calibration data
    file_name = 'wide_dist_pickle.p'
    parameters = pickle.load(open(file_name, 'rb'))
    mtx = parameters["mtx"]
    dist = parameters["dist"]
    print('mtx =', mtx)
    print('dist =', dist)

    #undistort the image
    undist = cv2.undistort(img, mtx, dist, None, mtx)
    print("undist np.shape(img):", np.shape(undist))

    # do perspective transform
    interim_result_persp = warper(undist,inv = False)

    # do edge detection
    edge_hls_img = pipeline_sobel_hls(interim_result_persp)

     # do the initial sliding window search
    binary_warped = edge_hls_img
    nonzero = binary_warped.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])
    out_img,histogram,left_lane_inds,right_lane_inds = slide_window(binary_warped)

    # Extract left and right line pixel positions
    leftx,lefty,rightx,righty = get_lane_pixels(binary_warped)

    # 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
    # result at this stage (after below) out_img is binary image with sliding windows
    # left lean is red , right lane is blue
    ploty = np.linspace(0, binary_warped.shape[0]-1, binary_warped.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]
    out_img[nonzeroy[left_lane_inds], nonzerox[left_lane_inds]] = [255, 0, 0]
    out_img[nonzeroy[right_lane_inds], nonzerox[right_lane_inds]] = [0, 0, 255]

    # Create an image to draw on and an image to show the selection window
    out_img = np.dstack((binary_warped, binary_warped, binary_warped))*255
    window_img = np.zeros_like(out_img)

    # Color in left and right line pixels
    out_img[nonzeroy[left_lane_inds], nonzerox[left_lane_inds]] = [255, 0, 0]
    out_img[nonzeroy[right_lane_inds], nonzerox[right_lane_inds]] = [0, 0, 255]

    # Generate a polygon to illustrate the search window area
    # Recast the x and y points into usable format for cv2.fillPoly()
    margin = 100 #TODO define margin in one spot
```

```
    left_line_window1 = np.array([np.transpose(np.vstack([left_fitx-margin, ploty]))])
    left_line_window2 = np.array([np.flipud(np.transpose(np.vstack([left_fitx+margin,
                              ploty])))])
    left_line_pts = np.hstack((left_line_window1, left_line_window2))
    right_line_window1 = np.array([np.transpose(np.vstack([right_fitx-margin, ploty]))])
    right_line_window2 = np.array([np.flipud(np.transpose(np.vstack([right_fitx+margin,
                                                                  ploty])))])
    right_line_pts = np.hstack((right_line_window1, right_line_window2))

    # Draw the lane onto the warped blank image
    # display isthe binary image, red left, blue right
    # green polygons showing region
    cv2.fillPoly(window_img, np.int_([left_line_pts]), (0,255, 0))
    cv2.fillPoly(window_img, np.int_([right_line_pts]), (0,255, 0))
    result = cv2.addWeighted(out_img, 1, window_img, 0.3, 0)

    final_result =  measure_curvature(edge_hls_img, undist,right_fitx,left_fitx, ploty)

    return final_result
```

Process an Image End to End

The section of this writeup that follow move step by setp through the functions used above to process an imnage. The disucsiosn begins begins with camera calibration and end with measuirng the radius of curvature and the lane offset.

Process Chessboard Images

The supplied images from the directory camera_cal are used in this step. After each image is converted to grayscale the OpenCV function `findChessboardCorners` is apppled to the chessboard images. The corners are the points where two black and two white squares intersect. The number of corners in a given row and a given columns for each calibration image are 9 (`nx`) and 6 (`ny`) respectively. The images and the corners are displayed for 500 ms each.

In [1]:
```
import numpy as np
import cv2
import glob
import matplotlib.pyplot as plt
# reads rgb
import matplotlib.image as mpimg

%matplotlib qt

# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*9,3), np.float32)
objp[:,:2] = np.mgrid[0:9,0:6].T.reshape(-1,2)

# 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.

# Make a list of calibration images
images = glob.glob('camera_cal/calibration*.jpg')

# Step through the list and search for chessboard corners
for fname in images:
    img = mpimg.imread(fname)
    gray = cv2.cvtColor(img,cv2.COLOR_RGB2GRAY)

    # Find the chessboard corners
    ret, corners = cv2.findChessboardCorners(gray, (9,6),None)

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

        # Draw and display the corners
        img = cv2.drawChessboardCorners(img, (9,6), corners, ret)
        cv2.imshow('img',img)
        cv2.waitKey(500)

cv2.destroyAllWindows()
```

Calibrate Camera

After the above cell runs the objpoints and imgpoints needed for camera calibration are availble. The corners coordinates of the chessboard, fount in the step above, are use to calibrate the camera. The cell below performs the actual calibration. The distortion coefficients and the camera matrix are found using OpenCV. These are applied on a test image image.

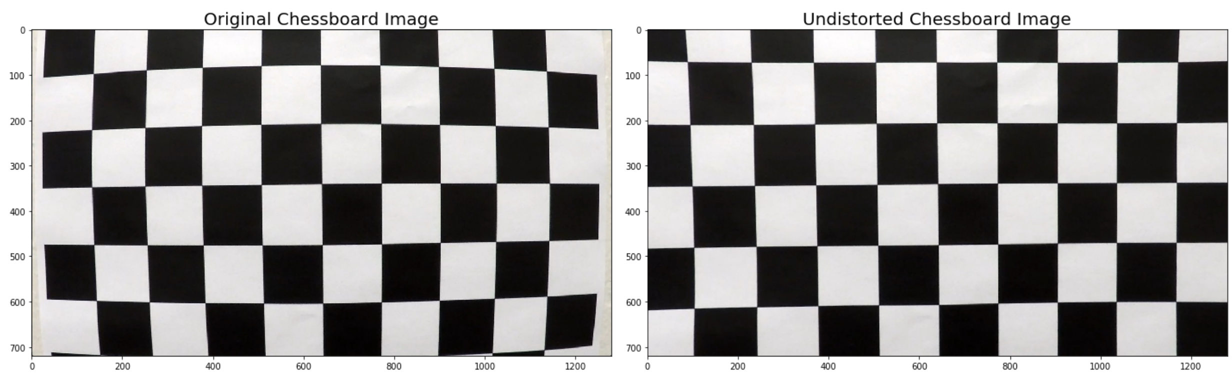
The key operation is `cv2.calibrateCamera(...)`. See the API here (http://docs.opencv.org/2.4.1/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html). This call returns the camera matrix `mtx`, a 3 x 3 matrix, and the distortion coefficents `dist`, a 1 x 5 matrix for this project. Note: The camera matrix and the distortion coeficients are saved in a pickle file for later use when processing the project video.

In [2]:

```
import pickle
%matplotlib inline

# Test undistortion on an image
img = mpimg.imread('camera_cal/calibration1.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,
img_size,None,None)


dst = cv2.undistort(img, mtx, dist, None, mtx)
cv2.imwrite('output_images/calibrated_image.jpg',dst)

# Save the camera calibration result for later use (we won't worry about rvecs / tvecs)
# this may not be striclty necessary for this project
dist_pickle = {}
dist_pickle["mtx"] = mtx
dist_pickle["dist"] = dist
pickle.dump( dist_pickle, open( "wide_dist_pickle.p", "wb" ) )
#dst = cv2.cvtColor(dst, cv2.COLOR_BGR2RGB)
# Visualize undistortion
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
f.tight_layout()
ax1.imshow(img)
ax1.set_title('Original Chessboard Image', fontsize=20)
ax2.imshow(dst)
ax2.set_title('Undistorted Chessboard Image', fontsize=20)
```

Out[2]:

```
<matplotlib.text.Text at 0x1b69f4a69b0>
```



The images above show one of the original chessboard images and an undistorted chessboard image.

Distortion Correction

Undistort the image using `cv2.undistort()` with the camera matrix `mtx` and the distortion coefficients `dist` on one of the test images.

In [4]:
```
#######################################
# Test distortion correction on a oad image
# Has the distortion correction been
# correctly applied to each image?
# Use one of the supplied test images
########################################
img = mpimg.imread('test_images/test2.jpg')
img_size = (img.shape[1], img.shape[0])
undist = cv2.undistort(img, mtx, dist, None, mtx)

# save image if desired
#mpimg.imsave('output_images/undist_image.jpg',undist)

# visualize undistortion on test image
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))
f.tight_layout()
ax1.imshow(img)
ax1.set_title('Original Test Image', fontsize=15)
ax2.imshow(undist)
ax2.set_title('Undistorted Test Image', fontsize=15)
```

Out[4]:
```
<matplotlib.text.Text at 0x1b69f810898>
```

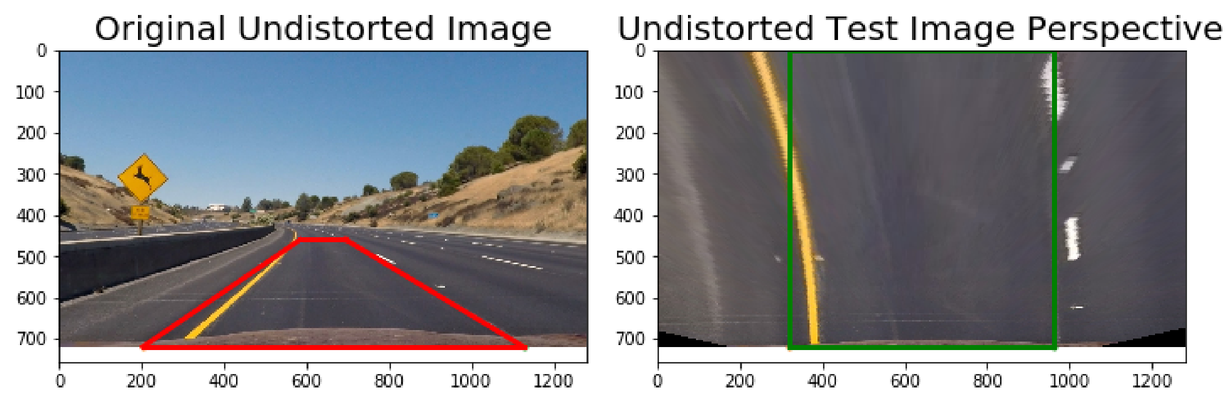
Apply the perspective transform function.

Apply the perspective transform on the undistorted image. Tre resulting images are shown following this code block. The source points are traced in red and the destination points are traced in green .

In [6]:

```
warpedPerspective = warper(undist,False)

# save the image if desired
# mpimg.imsave('output_images/warperPerspective.jpg',warpedPerspective)

# Visualize original undistorted test
# image and warped perspective

# src points
p1x = (img_size[0]/2 -55)
p1y = img_size[1]/2 + 100
p2x = (img_size[0]/6) -10
p2y = img_size[1]
p3x = (img_size[0] * 5 /6) + 60
p3y = img_size[1]
p4x = (img_size[0]/2 + 55)
p4y = img_size[1]/2 + 100

# destination points
d1x = (img_size[0]/4)
d1y = 0
d2x = (img_size[0]/4)
d2y = img_size[1]
d3x = (img_size[0]*3 /4)
d3y = img_size[1]
d4x = (img_size[0]*3 /4)
d4y = 0

# Visualize
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))
f.tight_layout()

#plot dots
ax1.imshow(undist)
ax1.plot(p1x,p1y,'.') # top right
ax1.plot(p2x,p2y,'.') # bottom right
ax1.plot(p3x,p3y,'.') # bottom left
ax1.plot(p4x,p4y,'.') # top left

# show outline of 4 src points
ax1.plot([p1x,p2x],[p1y,p2y],'red',lw=3)
ax1.plot([p3x,p4x], [p3y,p4y],'red', lw=3)
ax1.plot([p2x,p3x],[p2y,p3y], 'red', lw=3)
ax1.plot([p4x,p1x],[p4y,p1y], 'red', lw=3)
ax1.set_title('Original Undistorted Image', fontsize=20)

# destination
ax2.imshow(warpedPerspective)
ax2.plot(d1x,d1y,'.') # top right
ax2.plot(d2x,d2y,'.') # bottom right
ax2.plot(d3x,d3y,'.') # bottom left
ax2.plot(d4x,d4y,'.') # top left

# show outline of 4 dst points
ax2.plot([d1x,d2x],[d1y,d2y],'green',lw=3)
ax2.plot([d3x,d4x], [d3y,d4y],'green', lw=3)
ax2.plot([d2x,d3x],[d2y,d3y], 'green', lw=3)
ax2.plot([d4x,d1x],[d4y,d1y], 'green', lw=3)
ax2.set_title('Undistorted Test Image Perspective', fontsize=20)
```

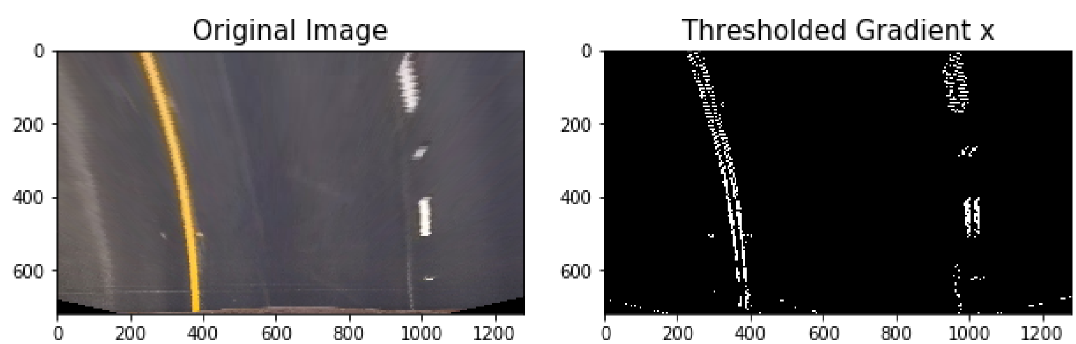
Out[6]: <matplotlib.text.Text at 0x1b6a1fe0a20>



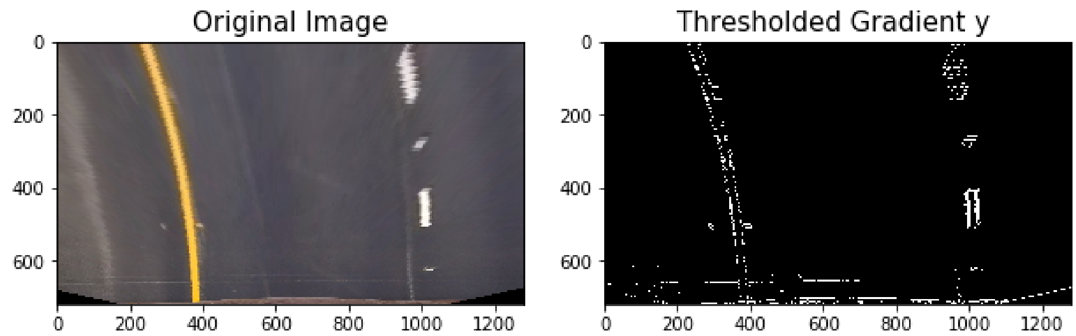
Test the Sobel function

The code below exercises the sobel function. The derivative is taken in the x direction and in the y direction. Although both pick up the lane line edges, the x derivative is a little cleaner.

In [14]:
```
# x direction
grad_binary = abs_sobel_thresh_phc(warpedPerspective, orient='x', thresh_min=25,
thresh_max=100)
# Plot the result
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
f.tight_layout()
ax1.imshow(warpedPerspective)
ax1.set_title('Original Image', fontsize=15)
ax2.imshow(grad_binary, cmap='gray')
mpimg.imsave('output_images/grad_binary.jpg',grad_binary, cmap='gray')
ax2.set_title('Thresholded Gradient x', fontsize=15)
plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)
```



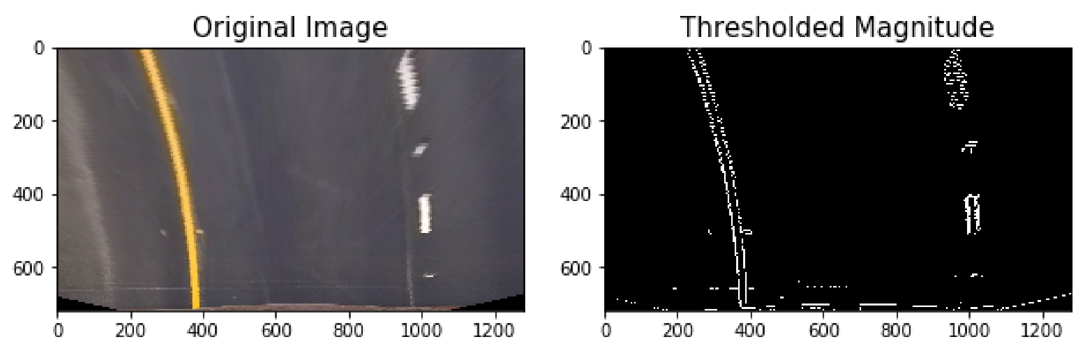
```
In [16]: # y direction
grad_binary_y = abs_sobel_thresh_phc(warpedPerspective, orient='y', thresh_min=20, thresh_m
ax=100)
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
f.tight_layout()
ax1.imshow(warpedPerspective)
ax1.set_title('Original Image', fontsize=15)
ax2.imshow(grad_binary_y, cmap='gray')
ax2.set_title('Thresholded Gradient y ', fontsize=15)
plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)
```



Magnitude of the Gradient

Exercise the magnitude of the magnitude of the gradient function. The lane lines are picked up nicely, other edges are present as well.

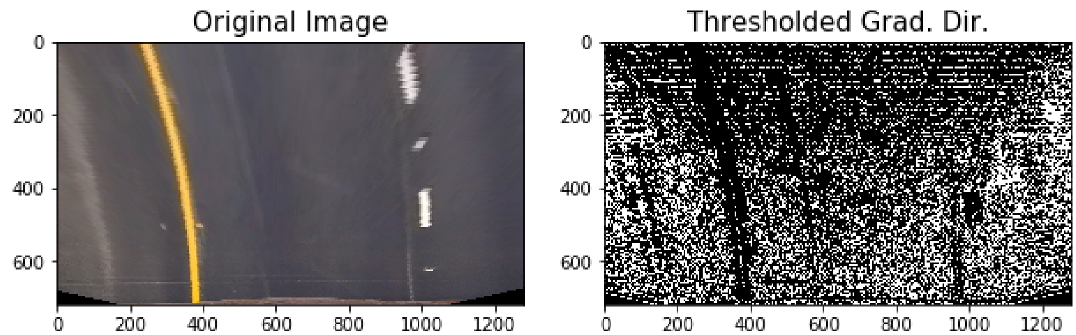
```
In [19]: # Run the function
mag_binary = mag_thresh(warpedPerspective, sobel_kernel=3, mag_thresh=(30, 100))
# Plot the result
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
f.tight_layout()
ax1.imshow(warpedPerspective)
ax1.set_title('Original Image', fontsize=15)
ax2.imshow(mag_binary, cmap='gray')
ax2.set_title('Thresholded Magnitude', fontsize=15)
plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)
```



Direction of the Gradient

Exercise the direction of the gradient function. Note the image is quite noisy.

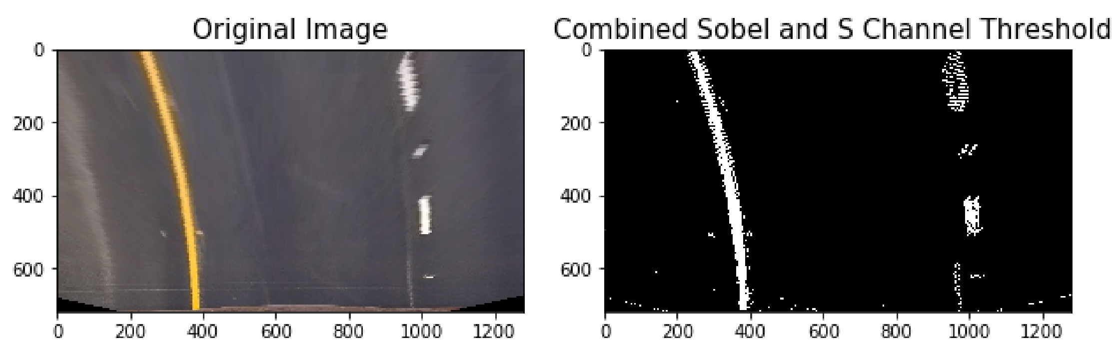
In [24]:
```
dir_binary = dir_threshold(warpedPerspective, sobel_kernel=15, thresh=(0.7, 1.3))

# Plot the result
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
f.tight_layout()
ax1.imshow(warpedPerspective)
ax1.set_title('Original Image', fontsize=15)
ax2.imshow(dir_binary, cmap='gray')
ax2.set_title('Thresholded Grad. Dir.', fontsize=15)
plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)
```



Combine Gradient and Color Channel Threshold

Combine the sobel gradient with a thresholded color channel. The S channel of a HLS image is thresolded and combined with a thresholded gradient. Of all of the techniques shown above, this approach does the best job of isolating the lane lines.

In [34]:
```
edge_hls_img= pipeline_sobel_hls(warpedPerspective)
# Plot the result
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
f.tight_layout()
ax1.imshow(warpedPerspective)
ax1.set_title('Original Image', fontsize=15)
ax2.imshow(edge_hls_img, cmap='gray')
ax2.set_title('Combined Sobel and S Channel Threshold.', fontsize=15)
plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)
```

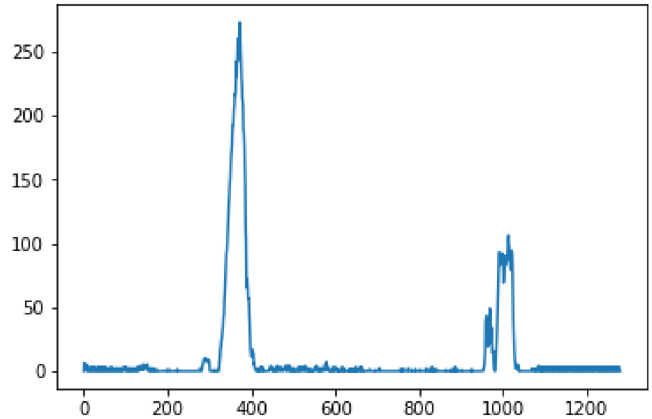


Exercise Sliding Window

The histogram for the thresholded image is shown below. As discsussed in the description of the `slide_window` function above, the histogram peaks are used as the basis for calculating the slding windows used to map the lane lines.

In [74]:
```
#plot the histogram
histogram = np.sum(edge_hls_img[edge_hls_img.shape[0]//2:,:], axis=0)
plt.plot(histogram)
```

Out[74]: [<matplotlib.lines.Line2D at 0x1b6a36dcdd8>]



In [52]:
```
# do the initial sliding window search
binary_warped = edge_hls_img
nonzero = binary_warped.nonzero()
nonzeroy = np.array(nonzero[0])
nonzerox = np.array(nonzero[1])
out_img,histogram,left_lane_inds,right_lane_inds = slide_window(binary_warped)

# Extract left and right line pixel positions
leftx,lefty,rightx,righty = get_lane_pixels(binary_warped)

# 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
# result at this stage (after below) out_img is binary image with sliding windows
# left lean is red , right lane is blue
ploty = np.linspace(0, binary_warped.shape[0]-1, binary_warped.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]
out_img[nonzeroy[left_lane_inds], nonzerox[left_lane_inds]] = [255, 0, 0]
out_img[nonzeroy[right_lane_inds], nonzerox[right_lane_inds]] = [0, 0, 255]

# Create an image to draw on and an image to show the selection window
out_img = np.dstack((binary_warped, binary_warped, binary_warped))*255
window_img = np.zeros_like(out_img)

# Color in left and right line pixels
out_img[nonzeroy[left_lane_inds], nonzerox[left_lane_inds]] = [255, 0, 0]
out_img[nonzeroy[right_lane_inds], nonzerox[right_lane_inds]] = [0, 0, 255]
```

```
Win: 640, 720, 272, 472, 913, 1113
Win: 560, 640, 282, 482, 884, 1084
Win: 480, 560, 274, 474, 884, 1084
Win: 400, 480, 263, 463, 907, 1107
Win: 320, 400, 254, 454, 904, 1104
Win: 240, 320, 241, 441, 904, 1104
Win: 160, 240, 223, 423, 891, 1091
Win: 80, 160, 202, 402, 864, 1064
Win: 0, 80, 179, 379, 864, 1064
```

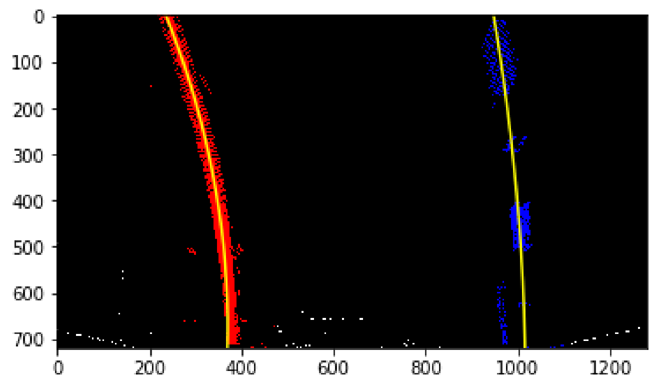
Window coordinates

The output above shows how the windows are calculated based on the histogram peaks. The first window of the left lane extends from y = 0 to y = 80. The corresponding x coordinates for the left lane, first window are 179, 379. The right lane window has the same y coordinates with x coordinates of 864 and 1064.

After finding all the lane pixels the left lane pixels are colored red and the right lane pixels are colored blue. The fitted polynomial is shown as a yellow line.

In [56]:
```
plt.imshow(out_img)
plt.plot(left_fitx, ploty, color='yellow')
plt.plot(right_fitx, ploty, color='yellow')
```

Out[56]: `[<matplotlib.lines.Line2D at 0x1b6a31c3c50>]`

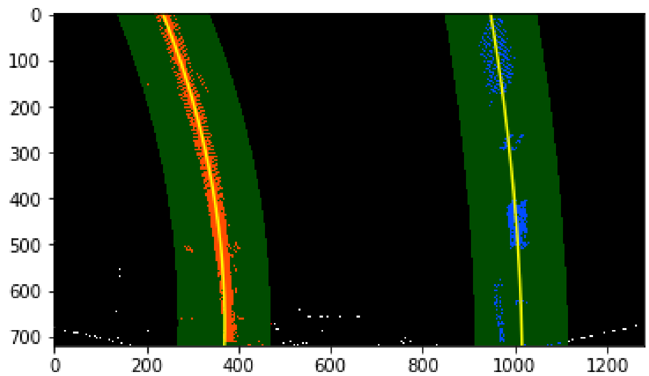


Search Region

A green polygon is shown around the computed lane lines.

In [58]:
```
# Generate a polygon to illustrate the search window area
# And recast the x and y points into usable format for cv2.fillPoly()

margin = 100 #TODO definbe maqrgin in one spot
left_line_window1 = np.array([np.transpose(np.vstack([left_fitx-margin, ploty]))])
left_line_window2 = np.array([np.flipud(np.transpose(np.vstack([left_fitx+margin,
                              ploty])))])
left_line_pts = np.hstack((left_line_window1, left_line_window2))
right_line_window1 = np.array([np.transpose(np.vstack([right_fitx-margin, ploty]))])
right_line_window2 = np.array([np.flipud(np.transpose(np.vstack([right_fitx+margin,
                                                                  ploty])))])
right_line_pts = np.hstack((right_line_window1, right_line_window2))

# Draw the lane onto the warped blank image
# display is  teh binary image, red left, blue right
# green polygons showing region
cv2.fillPoly(window_img, np.int_([left_line_pts]), (0,255, 0))
cv2.fillPoly(window_img, np.int_([right_line_pts]), (0,255, 0))
plt.plot(left_fitx, ploty, color='yellow')
plt.plot(right_fitx, ploty, color='yellow')
result = cv2.addWeighted(out_img, 1, window_img, 0.3, 0)
plt.imshow(result)
```

Out[58]: <matplotlib.image.AxesImage at 0x1b6a327f4a8>



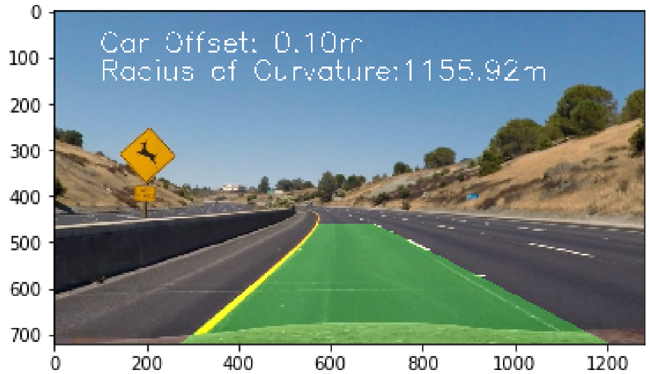
Measure Curvature

Final step

- Measure the radius of curvature
- Measure the car offset from the lane
- Perform and inverse transfrom to transfrom from teh cirs eye veiw to the roadway view

In [69]:
```
final_result =  measure_curvature(edge_hls_img, undist,right_fitx,left_fitx, ploty)
plt.imshow(final_result)
```

Out[69]: `<matplotlib.image.AxesImage at 0x1b6a374b908>`



The image above is the kind of image that is used in to make the video. The video is made by combining much of what had been shown individually in these steps, into a function call `process_image`. The `process_image` function is shwo above and is used with the code `proj4_clip = clip1.fl_image(process_image)` . 1261 images are processed from the supplied project video.

Discussion

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

- There are many things that I believe could be done to make this project more reobust. As it stands now I belive I have a minimal implementation. I was able to follow most of the material through about chapter 31 in class. In the last three or four chapters a large anount of new material was included without a great deal of explanation. I find that I am just staring to underastand some of the pixel lane finding code.
- My pipeline will likely fail in areas where there is a large amount of additioanl dark areas on the road such as shadows, tar stains, pot holes etc. It could be made more robust if more time was spent on thresholding, combining color channels, and perhaps additional gradient detection techniques. It may be useful to revisit Canny edge detection and see if it can be used or combined with some of the existing techniques.
- I had many challenges with the matplotlib in this project. I note that many other students had similar problems. Special thanks to the the forum members who tried to help. In many cases, the forum monitors advised a reinstall of several of the libraries. It was suggested that there may be some subtle bugs in matplotlib. I used Windows 10 and Anaconda.
- My project could be improved if I had more time. The semster is over and I still have to do project 5, so I am leaving off with this implementation.
- Some of the fucntions used above were not used in making the video. They are incldue for potential future use in project 5.
- All in all and interesting and challenging project. As indicated above it could be improved by adding more detail to the final chapters. In my be useful to add some material on OpenCV before directly using int in a major project.

In [ ]:
```
```