Advanced_Lane_Detection

April 2, 2019

1 Advanced Lane Finding

Brian Vlcek, April 2019

1.0.1 Through out this this notebook our goals will be to do the following tasks:

- Calibration of the forward facing camera found in the vehicle.
- Apply the correction to raw images found from the vehicle video stream.
- Apply color transforms, directional gradients, we will produce a much better lane detecting image.
- Apply a perspective transform the lane detected images to over head view images.
- We will generate a lane pixel detect and fit function.
- With the fits we can determine the curvature of the lane and vehicle position with respect to center.
- Finally we will warp the detected lane boundaries back onto the original image for the video stream with annotated information.

After each step is completed, I run a pipeline which completes each of these steps on a video stream, the result of which is found here.

```
In [1]: # import the required modules
        import numpy as np
        import cv2
        import glob
        import matplotlib.image as mpimg
        import matplotlib.pyplot as plt
        %matplotlib inline
```

1.0.2 Camera Calibration

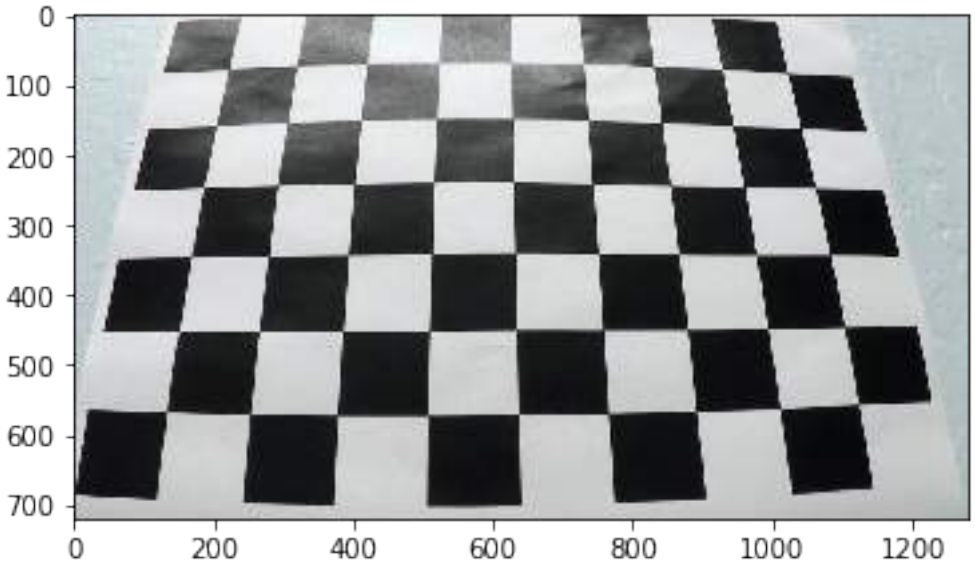
All cameras do not make perfect representations of the real world. There is always some source of noise when encoding the information about where objects are in the world into a 2-D image. Some of the major sources of error come from effects of the lens and CCD detection plane's orientation to the lens normal plane. OpenCV provides a way to compute how a standard set of real world objects with known properties (in our case a chessboard pattern) is transformed into an image a camera would generate by assuming the major factors are radial lens warping, sheer effects from non-perpendicular CCD surface to the lens housing, and the focusing of light by the lens. In order to attempt to correct these systematic errors, we provide OpenCV with a series of calibration

images taken with a single camera along with the known location of several points in the images in the 3D space it exists in. When feeding these images, and the real world location of points in the images to a function in OpenCV, an inversion of the distortion effects can be approximated and applied to any image taken with the same camera to give a better representation of what an ideal camera would generate. The steps to complete this calibration in OpenCV are done below.

In [2]:
```
# file paths to the calibration images
images_filepaths = glob.glob('camera_cal/calibration*.jpg')
print("there are {} calibration images".format(len(images_filepaths)))
```

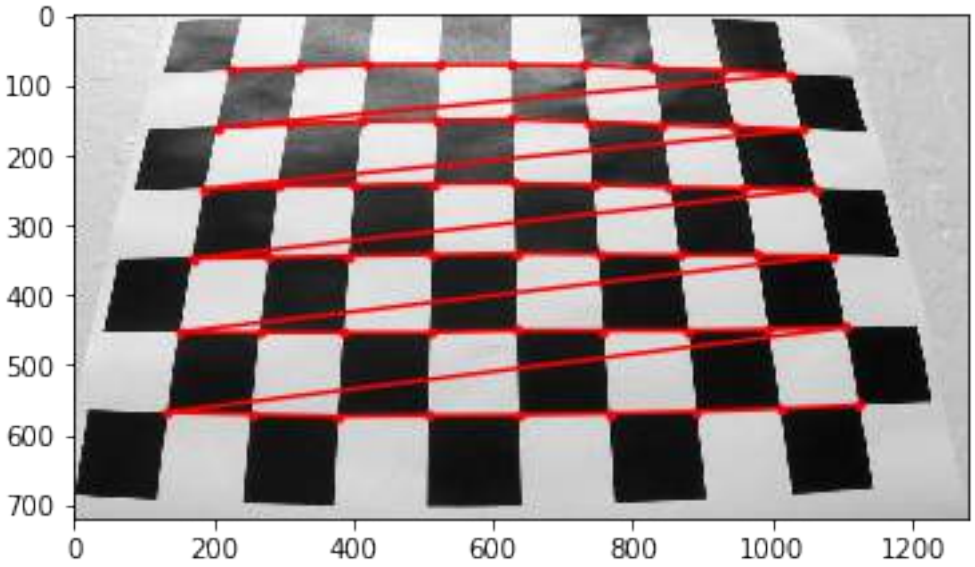
```
there are 20 calibration images
```

In [3]:
```
# lets take a look at one
img = cv2.imread(images_filepaths[5])
plt.imshow(img);
```



In [4]:
```
# so we've got a standard 9x6 chess board pattern
# we'll assume the object points are the intersection points
# of the chessboard pattern, as highlighted below
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
# coords are in (y,x) format
corners_foundQ,corners_coords= cv2.findChessboardCorners(gray, (9,6),None)
corners_coords = corners_coords.reshape(-1,2)
```

In [5]:
```
# lets draw the detected grid points
plt.imshow(gray,cmap='gray');
plt.plot(corners_coords.T[0],corners_coords.T[1],marker='.',c='r');
```



In [6]:
```
# now in order to map the detected corners to a real world object
# we need to know each corners location in a 3D space where the
# origin is place at the upper left corner and we image the true
# pattern is placed at z=0

real_objpts = np.zeros((6*9,3), np.float32)
real_objpts[:,:2] = np.mgrid[0:9,0:6].T.reshape(-1,2)
```

In [7]:
```
# some example points
real_objpts[:10]
```

Out[7]:
```
array([[ 0.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 2.,  0.,  0.],
       [ 3.,  0.,  0.],
       [ 4.,  0.,  0.],
       [ 5.,  0.,  0.],
       [ 6.,  0.,  0.],
       [ 7.,  0.,  0.],
       [ 8.,  0.,  0.],
       [ 0.,  1.,  0.]], dtype=float32)
```

In [8]:
```
# for each image we need the real world coordinates,
# we'll store the real world coordinates in a list
imgs_real_objpts = [] # 3d points in real world space
imgs_img_points = [] # 2d points in image plane.

# for each chessboard image see if you can
```

```
        # find al the corners, if so add it to the
        # lists in order to invert later.
        for image_file in images_filepaths:
            img = cv2.imread(image_file)
            gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
            # Find the chessboard corners
            corners_foundQ,corners_coords = cv2.findChessboardCorners(gray, (9,6),None)
            # If found, add object points, image points
            if corners_foundQ == True:
                imgs_real_objpts.append(real_objpts)
                imgs_img_points.append(corners_coords)
```

Now that we have stored detected points their 3D physical space points we can use the collection of those points to compute properties of the camera we are using.

```
In [9]: # Do camera calibration given object points and image points
        img_shape = (gray.shape[1],gray.shape[0])
        ret, camera_mtx, distortion_params, rvecs, tvecs = cv2.calibrateCamera(imgs_real_objpts,
```

The result of this calibration step follows the steps outlined in the paper "A Flexible New Technique for Camera Calibration" by Z Zhang (https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr98-71.pdf). The model is a series of matrices and vectors which models properties of the camera and the object being imaged. These properties are defined by a camera model such that any real world object given (like a point on the chessboard) by its coordinates in a base coordinate system $r_i = (X_i, Y_i, Z_i)$ are transformed to its pixel positions in the 2D image space given by (u_i, v_i). The transformation steps are done in a series, the first of which is,

$r'_i = R \cdot r_i + t$

where R is a rotation matrix intrinsic to the plane of the camera CCD and t a translation vector intrinsic to the placement of the CCD in the camera housing. After this is performed, the coordinates are scaled by the distance from the camera to give its projection on the CCD,

$r''_i = r'_i / z_i$

We then apply a transformation that models the effects of the distortion of the camera lens

$x'''_i = x''_i \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + 2p_1 x''_i y''_i + p_2(r^2 + 2x'^2)$

$y'''_i = x''_i \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + p_1(r^2 + 2(y''_i)^2) + 2p_2 x''_i y''_i$

where $r^2 = (x''_i)^2 + (y''_i)^2$ and the distortion coefficients are found in the variable dist. Finally the image is focused onto the CCD via passage through the lens giving.

$u_i = f_x x'''_i + c_x$

$v_i = f_y y'''_i + c_y$

these lens parameters are collected in the "camera matrix" stored in the variable mtx above.

```
In [10]: # distortion coefficients
         distortion_params
```

```
Out[10]: array([[-0.24688507, -0.02373155, -0.00109831,  0.00035107, -0.00259868]])
```

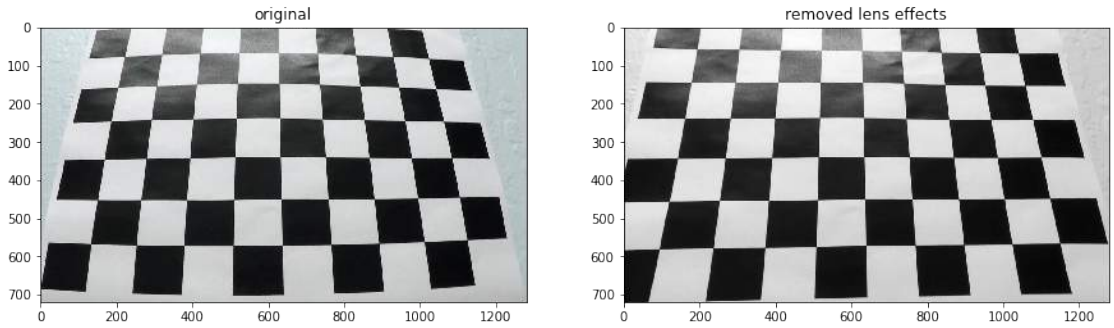
```
In [11]: # Camera Matrix
         camera_mtx
```

```
Out[11]: array([[  1.15777818e+03,   0.00000000e+00,   6.67113857e+02],
                [  0.00000000e+00,   1.15282217e+03,   3.86124583e+02],
                [  0.00000000e+00,   0.00000000e+00,   1.00000000e+00]])
```

We can invert the entire process for each image to produce images that would be generated from a nearly perfect camera by application of the function "undistort."

```
In [12]: img = cv2.imread(images_filepaths[5])
         gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
         ideal_img = cv2.undistort(gray, camera_mtx, distortion_params, None, camera_mtx)
```

```
In [13]: # take note of the rounded edges towards the outter edges of the images being corrected
         f, (ax1, ax2) = plt.subplots(1, 2, figsize=(15,7))
         ax1.imshow(img,cmap='gray')
         ax1.set_title('original', fontsize=12)
         ax2.imshow(ideal_img,cmap='gray')
         ax2.set_title('removed lens effects', fontsize=12);
```



1.0.3 Perspective Transform

After the lens/CCD effects are removed we have a nearly ideal image, but unfortunately we can't make too many useful measurements on the image when we're not looking straight on to the standardized object. In order to transform an image to a new perspective, we use OpenCV's tools to make a transformation of the image where reference points on the image are mapped to the orientation you would like. After the warping transformation is calculated, each pixel is mapped to the new orientation. When make this mapping you spread some of the pixels such that pixels that are next to each other in the original image are no longer neighbors in the final image. To fill in those pixels an interpolation method is used to make a good guess of the pixel values in the new perspective. The application of this is done below.

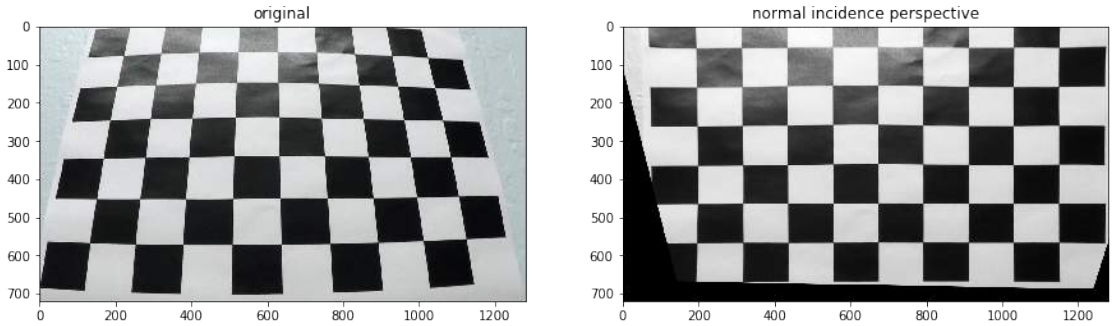
```
In [14]: # this is a special case function that works on chessboard patterns
         # with auto detection of chessboard points
         def normal_perspective_chessboard(raw_img, camera_mtx, distortion_coefs,nx=9,ny=6,recur
             if recursion == 10:
                 print('Unable to find corners at all')
                 warped = raw_img
```

```
            M = np.zeros(1)
            return warped,M

        undist = cv2.undistort(raw_img, camera_mtx, distortion_coefs, None, camera_mtx)
        gray = cv2.cvtColor(undist,cv2.COLOR_BGR2GRAY)
        cornersFound, corner_coords = cv2.findChessboardCorners(gray, (nx,ny), None)

        if cornersFound == True:
            print('({},{}) corners found'.format(nx,ny))
            # putting the cords into rows of (y,x) pairs
            src = corner_coords.reshape(-1,nx,2)

            UL = src[0,0] #img upper left
            UR = src[0,-1]#img upper right
            LL = src[-1,0] #img lower left
            LR = src[-1,-1] #img lower right

            src = np.float32([UL,UR,LL,LR])

            ulx,uly = UL
            lrx,lry = LR
            # destination cordinates is perfect rectangle
            dst = np.float32([[ulx,uly],[lrx,uly],[ulx,lry],[lrx,lry]])
            # get the linear warp matrix
            M = cv2.getPerspectiveTransform(src,dst)
            #apply warp matrix
            warped = cv2.warpPerspective(gray,M,gray.shape[::-1], flags=cv2.INTER_LINEAR)
        else:
            print('chessboard corners could not automatically be found reducing nx')
            warped,M=normal_perspective_chessboard(raw_img, camera_mtx, distortion_coefs,nx

        return warped, M
```

```
In [15]: new_perspective,warp_mtx = normal_perspective_chessboard(img,camera_mtx,distortion_para
```

```
(9,6) corners found
```

```
In [16]: f, (ax1, ax2) = plt.subplots(1, 2, figsize=(15,7))
         ax1.imshow(img,cmap='gray')
         ax1.set_title('original', fontsize=12)
         ax2.imshow(new_perspective,cmap='gray')
         ax2.set_title('normal incidence perspective', fontsize=12);
```



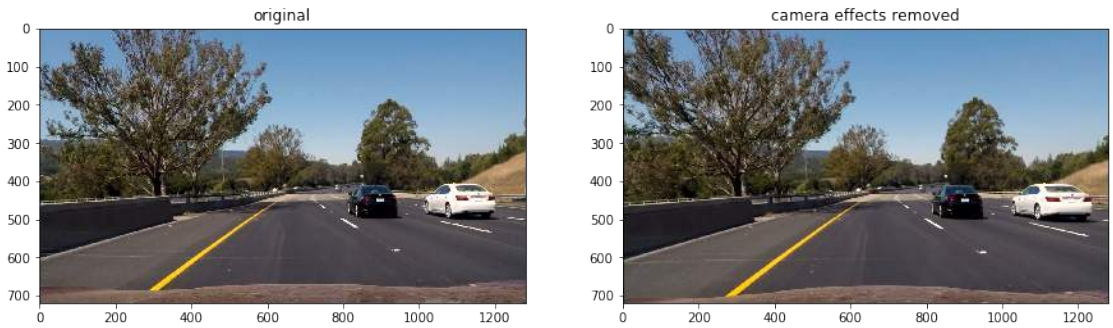
1.0.4 Applying Correction to Road Images

Here we're going use the calibration transform to correct images captured by our car mounted camera. After we correct for lens and CCD effects.

In [17]:
```
# file paths to the road images
road_image_paths = glob.glob('test_images/*.jpg')
print("there are {} test road images".format(len(road_image_paths)))
```

```
there are 8 test road images
```

In [18]:
```
img = mpimg.imread('test_images/test6.jpg')
ideal_img = cv2.undistort(img, camera_mtx, distortion_params, None, camera_mtx)


# notice how the size of the rear bumper of the white car is much larger
# than in the raw image
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(15,7))
ax1.imshow(img,cmap='gray')
ax1.set_title('original', fontsize=12)
ax2.imshow(ideal_img,cmap='gray')
ax2.set_title('camera effects removed', fontsize=12);
```



Take notice the size of the cars towards the outer edges of the image. You can see the white car is much larger than it appears in the unaltered image.

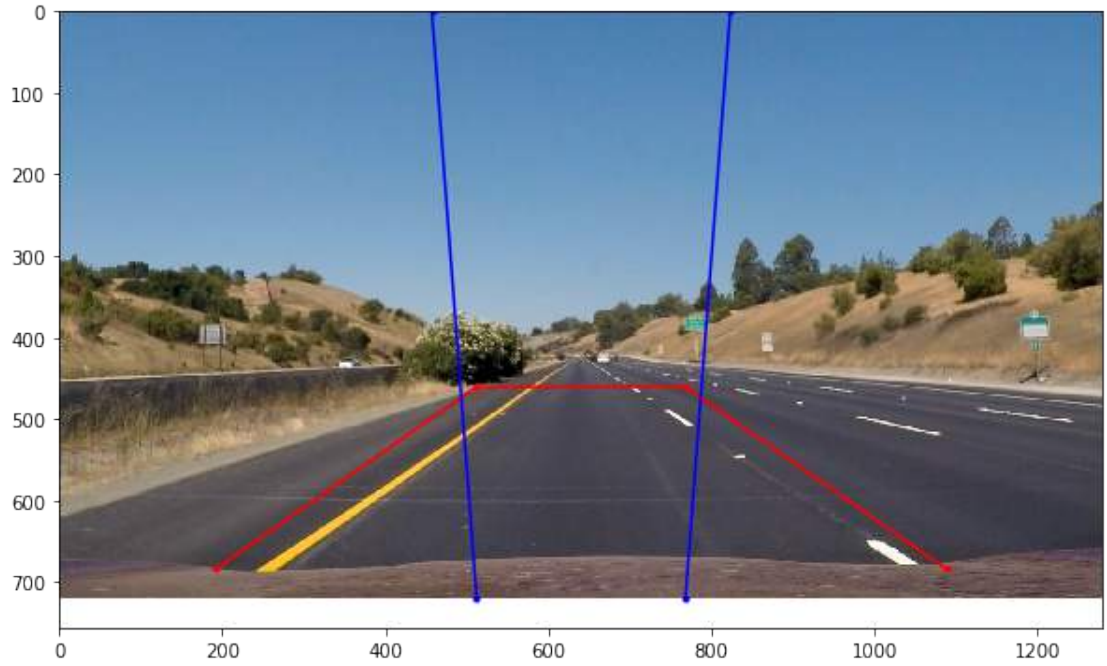
1.0.5 Perspective Change to the Birds Eye View

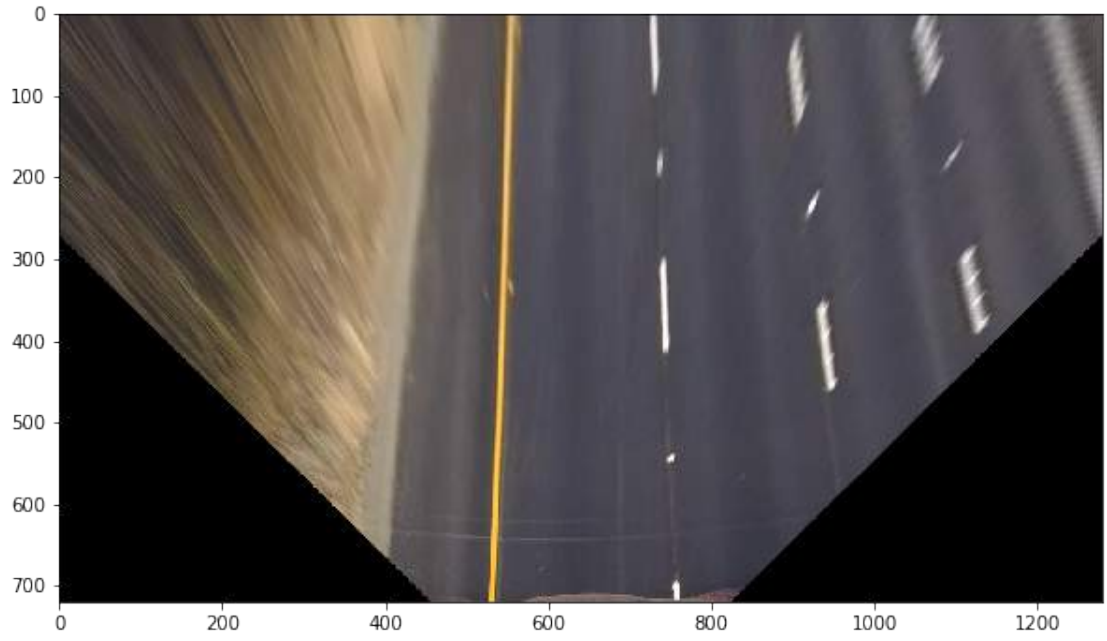
Now that we've been able to undistort images, we want to change the perspective of the image such that we have a birds eye view (BEV) of the lane. This can be done by warping the image in the same way we re-oriented the chessboard earlier. Of course this is done AFTER applying the calibration step. An example of this perspective change is done below.

```
In [19]: img = mpimg.imread('test_images/straight_lines1.jpg')
         ydim,xdim,_ = img.shape

         #undistort image
         undistort_lanes = cv2.undistort(img, camera_mtx, distortion_params, None, camera_mtx)

         # lets set our origin points to be warped
         src = np.float32([[0.15*xdim,0.95*ydim],#lower left
                           [0.4*xdim,0.64*ydim], #upper left
                           [0.6*xdim,0.64*ydim], #upper right
                           [0.85*xdim,0.95*ydim]])#lower right

         # location to warp them to (destination points)
         ulx,uly = 0.4*xdim,0.0*ydim
         lrx,lry = 0.6*xdim,1.0*ydim
         dst = np.float32([[ulx,lry],[ulx-55,uly],[lrx+55,uly],[lrx,lry]])

         # Applying the Warp
         warp_M = cv2.getPerspectiveTransform(src,dst)
         inverse_warp_M = cv2.getPerspectiveTransform(dst,src)
         BEV_lanes = cv2.warpPerspective(img, warp_M, img[:,:,0].shape[::-1], flags=cv2.INTER_LI
```

The image below shows in red the source points and where we want to warp them to after the application of the perspective transform. Notice that I have to spread out the lane points that are converging, such that the goal of this particular image is to generate parallel lane lines.

```
In [20]: plt.figure(figsize=(10,7))
         plt.imshow(img,cmap='gray')
         plt.plot(src.T[0],src.T[1],marker='.',c='r');
         plt.plot(dst.T[0],dst.T[1],marker='.',c='b');

         plt.figure(figsize=(10,7))
         plt.imshow(BEV_lanes);
```





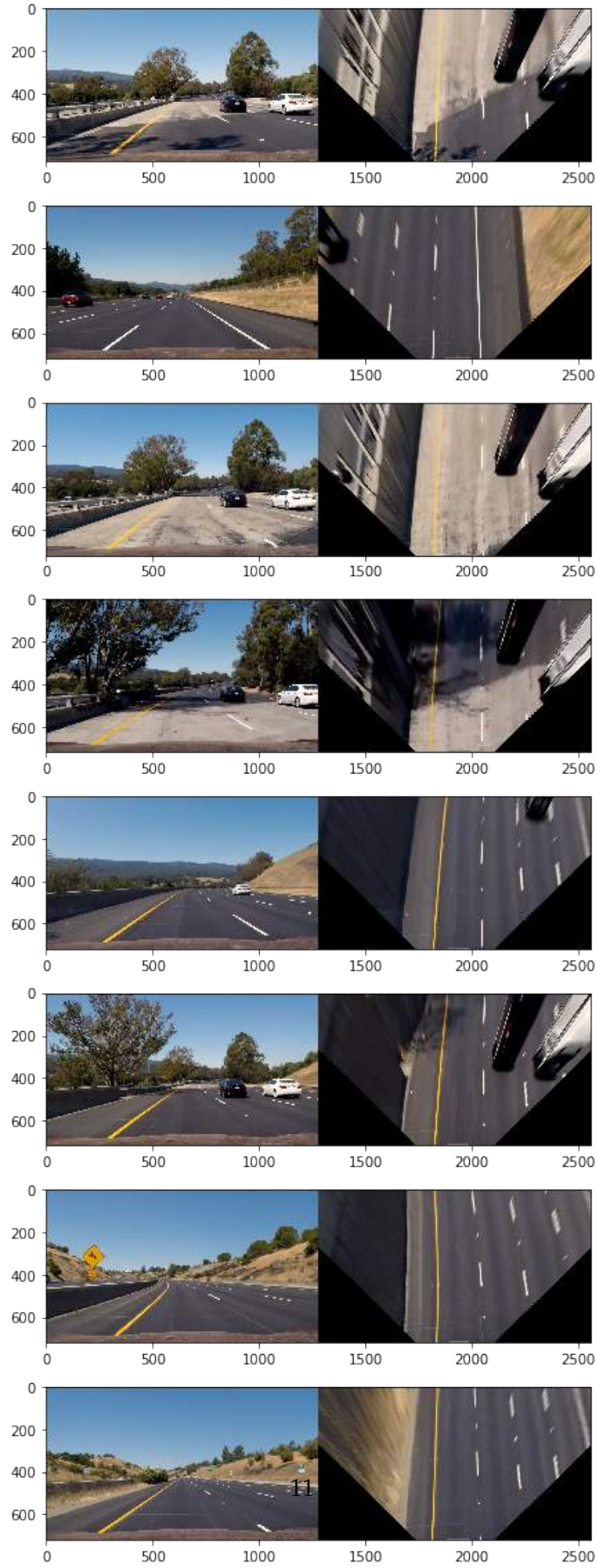
lets wrap this up in a function for later

```
In [21]: def warp_to_BEV(lanes_rgb_img,camera_mtx,distort_coefs):
             undistorted_lanes = cv2.undistort(lanes_rgb_img, camera_mtx, distort_coefs, None, c
```

```
    ydim,xdim,_ = undistorted_lanes.shape
    # lets set our warp points
    src = np.float32([
                [0.15*xdim,0.95*ydim],#lower left
                [0.4*xdim,0.64*ydim], #upper left
                [0.6*xdim,0.64*ydim], #upper right
                [0.85*xdim,0.95*ydim]])#lower right

    # destination points
    ulx,uly = 0.4*xdim,0.0*ydim
    lrx,lry = 0.6*xdim,1.0*ydim

    # destination cordinates
    dst = np.float32([[ulx,lry],[ulx-55,uly],[lrx+55,uly],[lrx,lry]])

    warp_M = cv2.getPerspectiveTransform(src,dst)
    inverse_warp_M = cv2.getPerspectiveTransform(dst,src)
    BEV_lanes = cv2.warpPerspective(undistorted_lanes, warp_M, undistorted_lanes[:,:,0]

    return((BEV_lanes,inverse_warp_M))
```

Through out this notebook, I will check parameter values and functions against all the test images, to make sure that the choices tend to generalize to the test case set well.

```
In [22]: f, axs = plt.subplots(len(road_image_paths),1,figsize=(30,16))
         f.tight_layout()

         for idx in range(len(road_image_paths)):
             current_img = mpimg.imread(road_image_paths[idx])
             warped_rgb,inverse_warp_mtx = warp_to_BEV(current_img,camera_mtx, distortion_params
             current_img = np.hstack((current_img,warped_rgb))
             axs[idx].imshow(current_img)
```


0
200
400
600
0
500
1000
1500
2000
2500

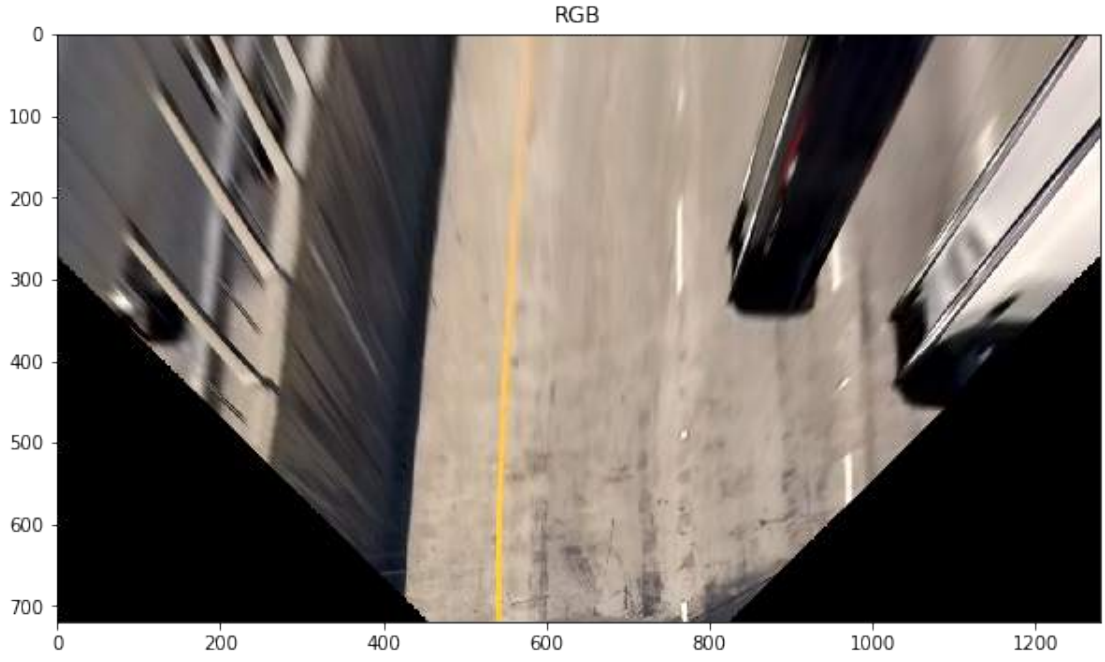
1.0.6 Color Transforms and Directional Gradients

Ultimately our goal is to be able to track the lane lines much better than our previous attempt done with Hough lines. In order to do that we'll want to apply some function on the BEV images such that we highlight the lane lines in all lighting and background road cases.

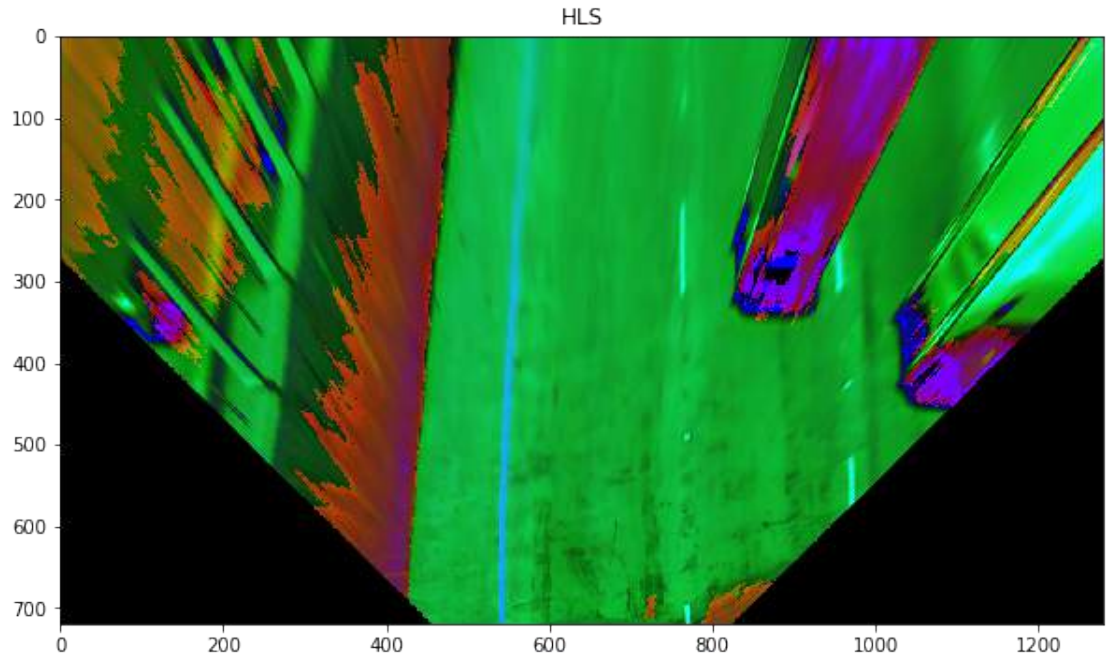
The major challenge of being able to detect lane lines from camera data is that the lighting conditions can vary wildly as you travel down the road. What we need is some feature of the lane that remains constant in most lighting conditions. To find these constant features we need to look at color spaces. All images can be decomposed into RGB channels but this is not the only color space basis. We can use HSV,HSL, or even LAB color spaces to represent the image as well. The major advantage of these alternative color spaces is brightness and base colors (like dark red or light red have a common base color of red) are the channels rather than just RGB colors. Let's take a look at the road way images in these other color spaces to see some valuable information come to light.

In [23]:
```
current_img = mpimg.imread(road_image_paths[2])
warped_rgb,inverse_warp_mtx = warp_to_BEV(current_img,camera_mtx, distortion_params)

# transforming to the other color space
hls = cv2.cvtColor(warped_rgb, cv2.COLOR_RGB2HLS)
hsv = cv2.cvtColor(warped_rgb, cv2.COLOR_RGB2HSV)
lab = cv2.cvtColor(warped_rgb, cv2.COLOR_RGB2LAB)
```

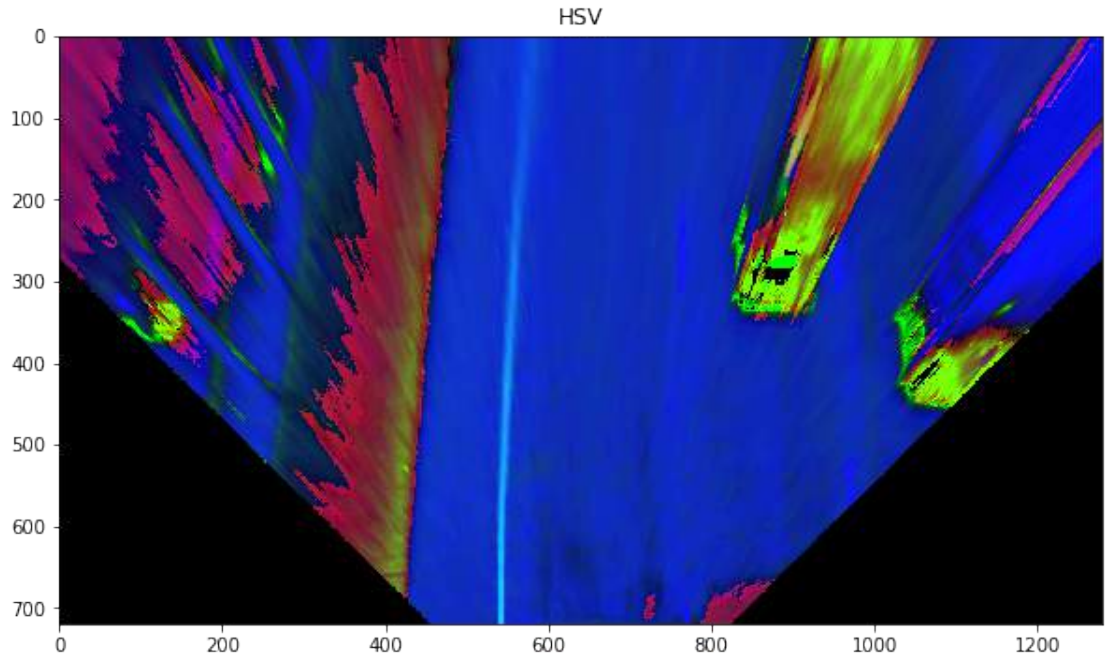
In [24]:
```
# Plot the colorspaces
plt.figure(figsize=(10, 9))
plt.title('RGB', fontsize=12)
plt.imshow(warped_rgb);
```



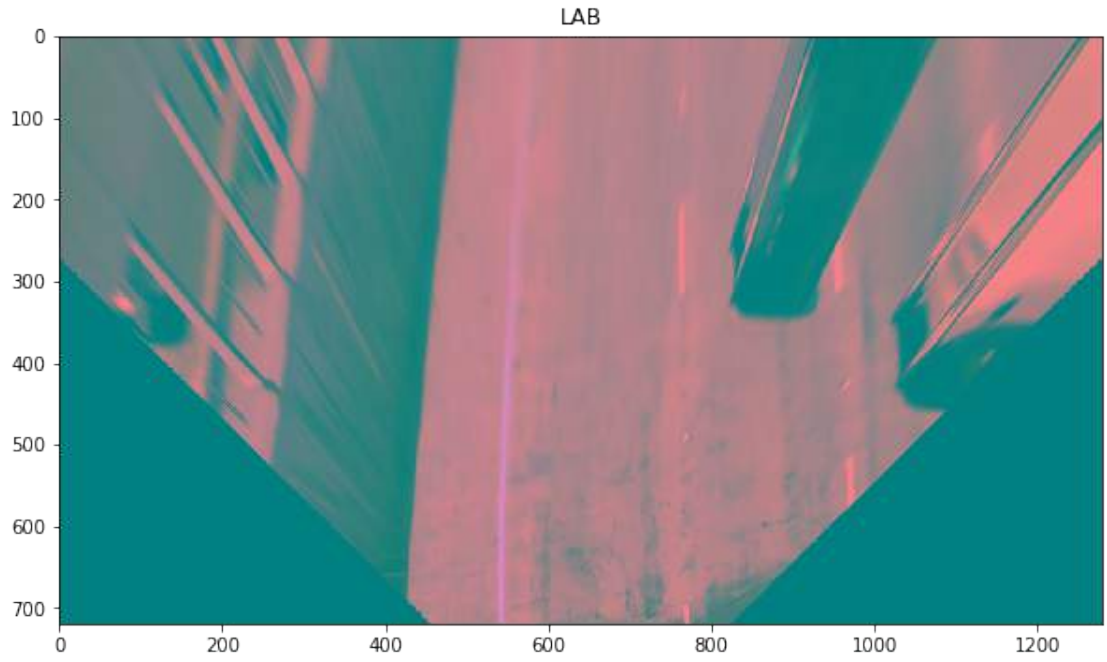
In [25]:
```
# Plot the result
plt.figure(figsize=(10, 9))
plt.title('HLS', fontsize=12)
plt.imshow(hls);
```



In [26]:
```
# Plot the result
plt.figure(figsize=(10, 9))
plt.title('HSV', fontsize=12)
plt.imshow(hsv);
```



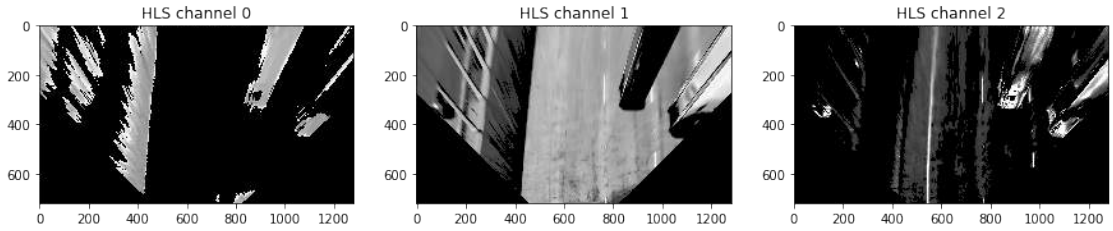
In [27]:
```
# Plot the result
plt.figure(figsize=(10, 9))
plt.title('LAB', fontsize=12)
plt.imshow(lab);
```



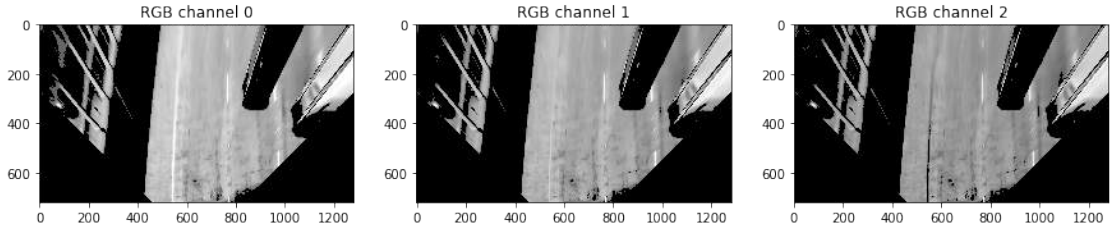
We can see that in the HSV, HSL, and especially the LAB color spaces, the yellow lane marker is highlighted within the shadows of the image or not! Let explore this further by looking at applying thresholding to the individual channels.

```
In [28]: def color_space_thresh(img, thresh=(0, 255)):
             tmp = np.copy(img)
             # 3 channel images
             if len(img.shape) == 3:
                 for ch in range(3):
                     tmp[~((img[:,:,ch]>thresh[0]) & (img[:,:,ch]<=thresh[1]))] = 0
             else:
                 # binary images
                 tmp[~((img>thresh[0]) & (img<=thresh[1]))] = 0

             return tmp
```

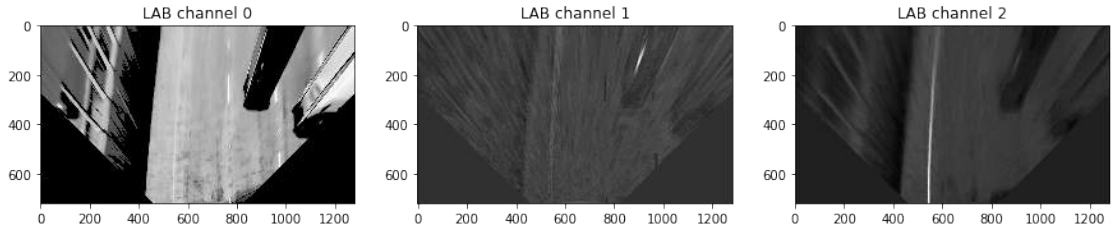
```
In [29]: # applying thresholds on the color channels for HLS space
         threshelds = [color_space_thresh(hls[:,:,ch],(50,255)) for ch in range(3)]

         f, axs = plt.subplots(1, 3, figsize=(15,7))
         for idx in range(len(axs)):
             axs[idx].imshow(threshelds[idx],cmap='gray')
             axs[idx].set_title('HLS channel '+str(idx), fontsize=12);
```




```
In [30]: # applying thresholds on the color channels for RGB space
         threshelds = [color_space_thresh(warped_rgb[:,:,ch],(100,255)) for ch in range(3)]

         f, axs = plt.subplots(1, 3, figsize=(15,7))
         for idx in range(len(axs)):
             axs[idx].imshow(threshelds[idx],cmap='gray')
             axs[idx].set_title('RGB channel '+str(idx), fontsize=12);
```



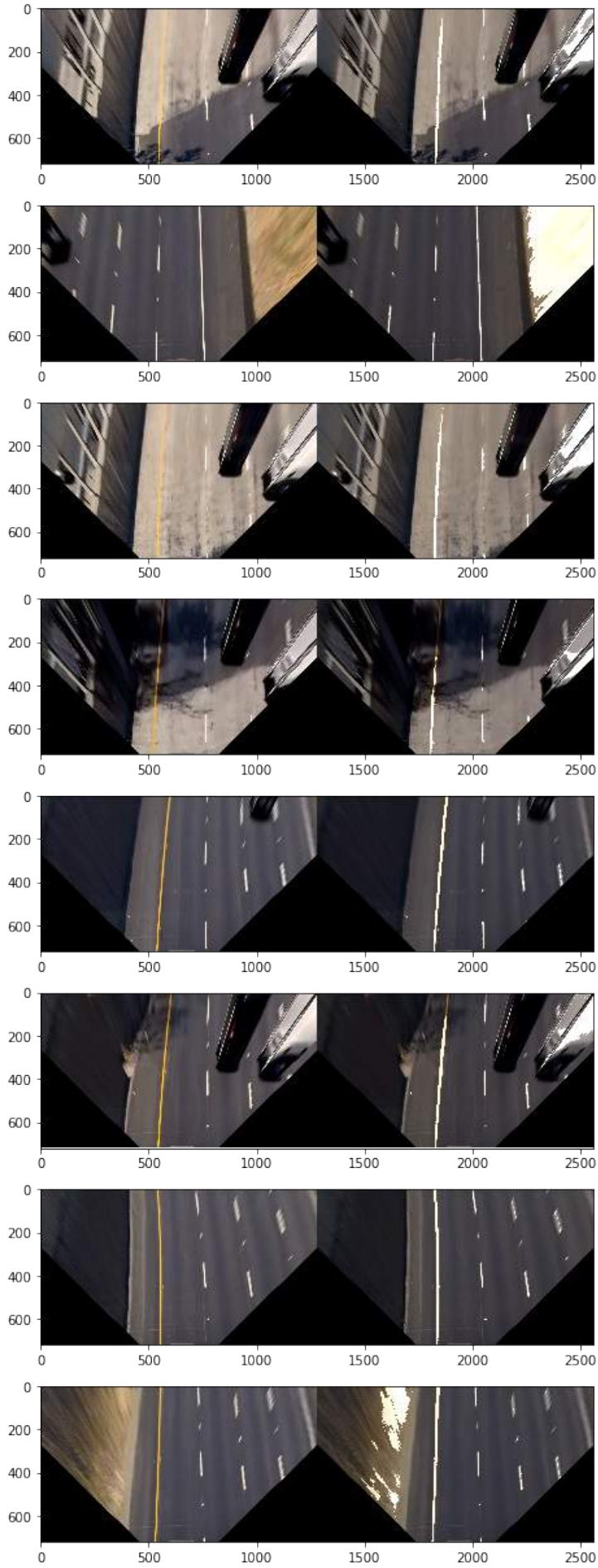
```
In [31]: # applying thresholds on the color channels for HLS space
         threshelds = [color_space_thresh(lab[:,:,ch],(70,255)) for ch in range(3)]

         f, axs = plt.subplots(1, 3, figsize=(15,7))
         for idx in range(len(axs)):
             axs[idx].imshow(threshelds[idx],cmap='gray')
             axs[idx].set_title('LAB channel '+str(idx), fontsize=12);
```



Looks like the B-channel of the LAB image makes a very good way of detecting lane lines regardless of shadows or direct lighting. We can use the RGB image with high thresholds across all channels to select white colors to highlight the white lane markers. Let's apply high thresholds to RGB channels to select the white components of the image, and a threshold on the B channel of the LAB color space to select the yellow lanes in a variety of lighting conditions. After we make these thresholds we want to overlay the selected pixels over the original RGB image only slightly darkened to really increase the contrast between lane lines and the background. Let's see this applied below.

```
In [32]: def select_yellows_whites(warped_RGB,b_thresh=(150,255),rgb_thresh=(200,255)):
             # thresholding to only grab near white pixels
             whites_img = color_space_thresh(warped_RGB,rgb_thresh)

             # selecting the LAB,B-ch
             lab_img = cv2.cvtColor(warped_RGB, cv2.COLOR_RGB2LAB)
             b_ch = lab_img[:,:,2]
             b_img = color_space_thresh(b_ch,b_thresh)

             b_bin_img = np.dstack((b_img,b_img,b_img))

             combined = cv2.addWeighted(whites_img,1.0,b_bin_img,1.0,0.0)

             return combined

In [33]: def hilight_lanes(warped_rgb,b_thresh=(150,255),rgb_thresh=(200,255)):
             yw_select = select_yellows_whites(warped_rgb,b_thresh=(150,255),rgb_thresh=(200,255
             hilight = cv2.addWeighted(warped_rgb,0.8,yw_select,1.0,0.0)
             return hilight
```

In [34]:
```
f, axs = plt.subplots(len(road_image_paths),1,figsize=(26,17))
f.tight_layout()

for idx in range(len(road_image_paths)):
    current_img = mpimg.imread(road_image_paths[idx])
    warped_rgb,inverse_warp_mtx = warp_to_BEV(current_img, camera_mtx, distortion_param
    hilight = hilight_lanes(warped_rgb)

    combined = np.hstack((warped_rgb,hilight))
    axs[idx].imshow(combined)
```


0
200
400
600
0
500
1000
1500
2000
2500

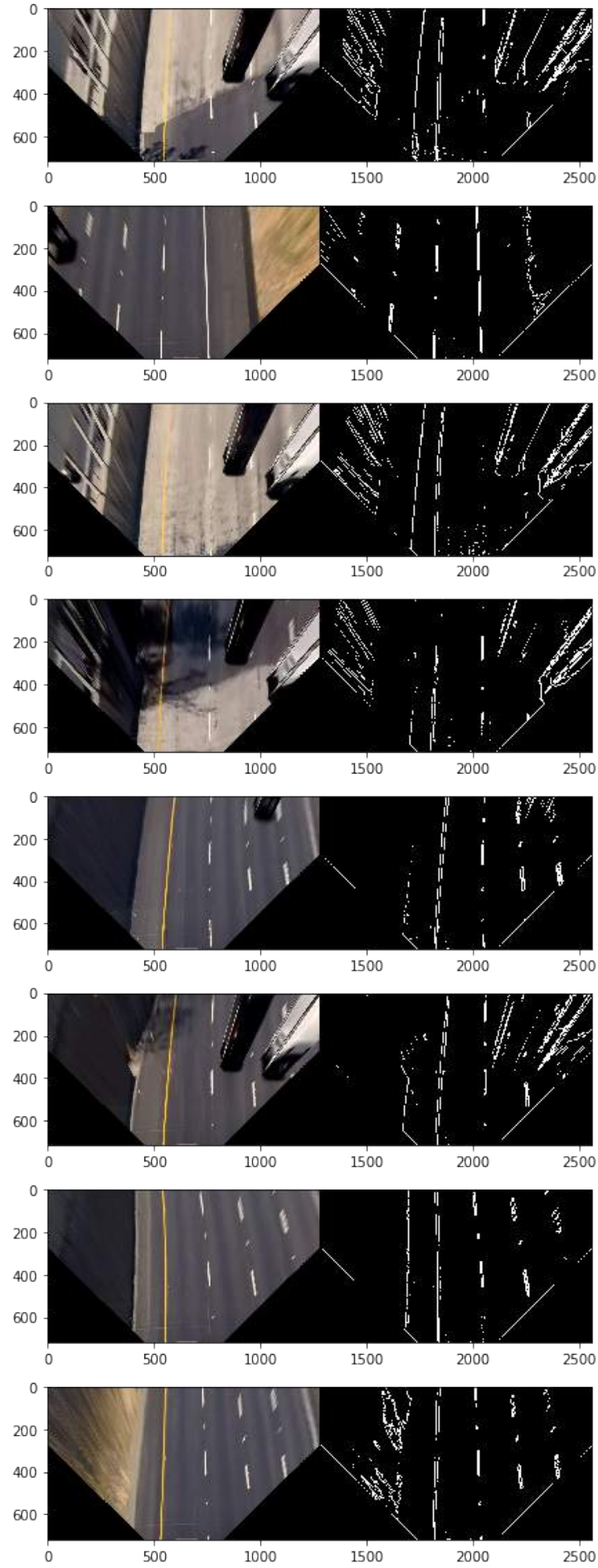
Notice now that our lane markers are much brighter in most cases that on the original RGB images. In order to ONLY select the lane pixels we will apply a gradient application to the highlited image where we'll select the directions of the gradients to favor the horizontal direction vs. the vertical. This will filter out any edges you find that are horizontal, like road material type transitions, rather than vertical edges like the lane markers in the images above.

```
In [35]: # function applys a directional gradient over the entire image
         # thresholding is then applied to the direction angle and magintude of the gradient
         def grad_threshold(img,gauss_kernel_width=None,gray_convert=cv2.COLOR_RGB2GRAY,\
                            sobel_kernel=3, mag_thresh=(0,255),\
                            angle_thresh=[[0, 180*np.pi/180],[180*np.pi/180,360*np.pi/180.],[0.0

             if len(img.shape) == 2:
                 gray = img
             else:
                 # if its not a bin image use the gray-scale version
                 gray = cv2.cvtColor(img,gray_convert)

             # option to apply de-noising if you wanted to
             if gauss_kernel_width != None:
                 gray = cv2.GaussianBlur(gray,(gauss_kernel_width,gauss_kernel_width),0)

             xgrad = cv2.Sobel(gray, cv2.CV_64F, 1, 0,ksize=sobel_kernel)
             ygrad = cv2.Sobel(gray, cv2.CV_64F, 0, 1,ksize=sobel_kernel)
             angle = np.arctan2(ygrad,xgrad)

             # convert angles to 0->2pi
             angle[(angle < 0.0)] = 2.*np.pi+angle[(angle < 0.0)]

             # direction of gradient threshold (to have 2 ranges i need 3 masks)
             angle_mask1 = np.zeros_like(gray)
             angle_mask1[(angle >= angle_thresh[0][0]) & (angle <= angle_thresh[0][1])] = 1

             angle_mask2 = np.zeros_like(gray)
             angle_mask2[(angle >= angle_thresh[1][0]) & (angle <= angle_thresh[1][1])] = 1

             angle_mask3 = np.zeros_like(gray)
             angle_mask3[(angle >= angle_thresh[2][0]) & (angle <= angle_thresh[2][1])] = 1


             # magnitude gradient thresholding
             gradmag = np.sqrt( xgrad**2 + ygrad**2 )
             grad8bit = np.uint8(255*gradmag/np.max(gradmag))

             mag_mask = np.zeros_like(grad8bit)
             mag_mask[(grad8bit >= mag_thresh[0]) & (grad8bit <= mag_thresh[1])] = 1
```

```
        # apply logical or on the angles combined with a logical and on the magnitude
        combined = np.zeros_like(gray)
        combined[((angle_mask1 == 1) | (angle_mask2 == 1) | (angle_mask3==1)) & (mag_mask =

        return combined
```

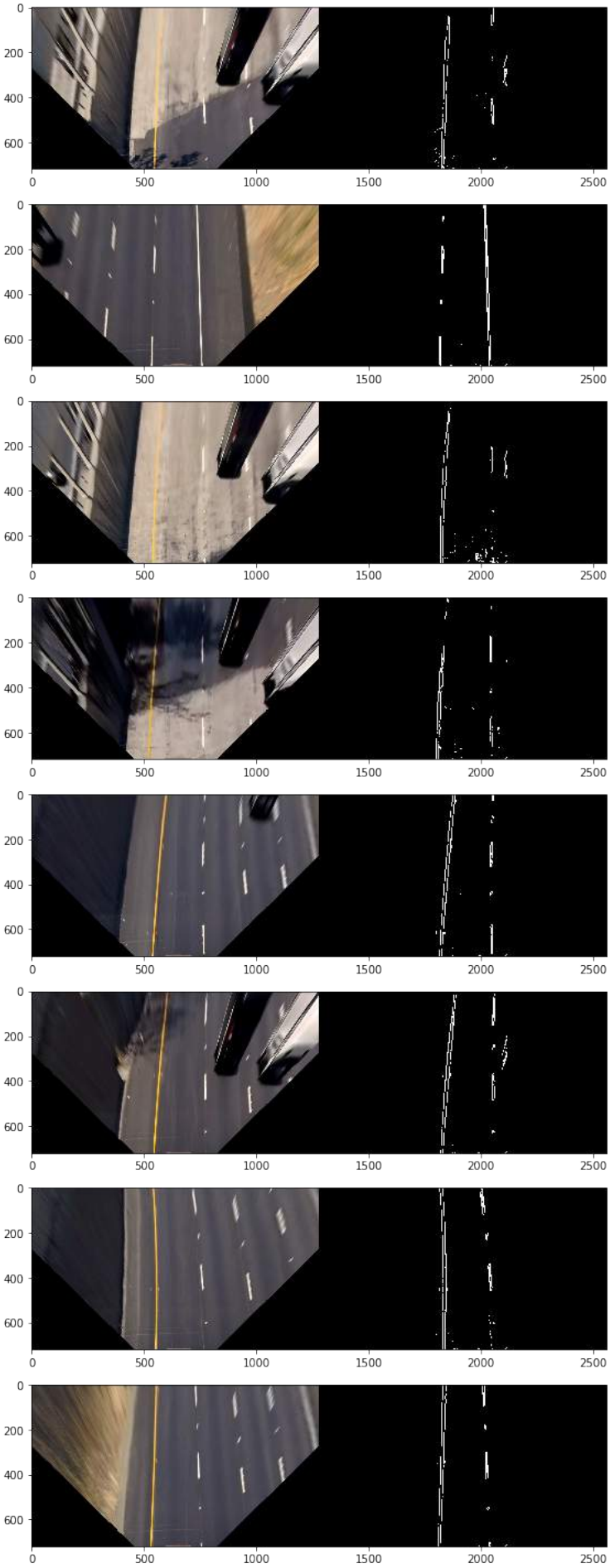
In [36]:
```
def hilight_and_edge(warped_RGB,b_thresh=(150,255),rgb_thresh=(200,255),
                     alpha=70.,sobel_kernel=5,grad_mag_thresh=(25,255)):

    Hilight = hilight_lanes(warped_RGB,b_thresh=b_thresh,rgb_thresh=rgb_thresh)
    G_img = grad_threshold(Hilight,
                           sobel_kernel=sobel_kernel,
                           mag_thresh=grad_mag_thresh,
                           angle_thresh=[[0.*np.pi/180., alpha*np.pi/180.],
                                     [(180.-alpha)*np.pi/180, (180.+alpha)*np.pi/180.],
                                     [(360.-alpha)*np.pi/180.,360.*np.pi/180.]])
    return G_img
```

In [37]:
```
# for each test image show the BEV lane edges
f, axs = plt.subplots(len(road_image_paths),1,figsize=(30,16))
f.tight_layout()

for idx in range(len(road_image_paths)):
    current_img = mpimg.imread(road_image_paths[idx])
    warped_rgb,inverse_warp_mtx = warp_to_BEV(current_img,camera_mtx, distortion_params
    g_img = hilight_and_edge(warped_rgb,alpha=80.)
    g_img = np.dstack(np.array((g_img,g_img,g_img))*255)

    current_img = np.hstack((warped_rgb,g_img))
    axs[idx].imshow(current_img)
```


0
200
400
600
0
500
1000
1500
2000
2500

Excellent. It looks like if we apply a region of interest mask on these images we will be able to easily isolate the lane pixels in similar conditions as the test images.

```
In [38]: def region_of_interest(img, vertices):
             #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 depending on the inp
             if len(img.shape) > 2:
                 channel_count = img.shape[2]  # i.e. 3 or 4 depending on your image
                 ignore_mask_color = (255,) * channel_count
             else:
                 ignore_mask_color = 255

             #filling pixels inside the polygon defined by "vertices" with the fill color
             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 [39]: # applying the gradient thresholds to each image
         f, axs = plt.subplots(len(road_image_paths),1,figsize=(10,20))
         f.tight_layout()

         for idx in range(len(road_image_paths)):
             current_img = mpimg.imread(road_image_paths[idx])
             warped_rgb,inverse_warp_mtx = warp_to_BEV(current_img,camera_mtx, distortion_params
             g_img = hilight_and_edge(warped_rgb,alpha=80.)

             # building the mask
             ydim,xdim,_ = current_img.shape
             roi_mask = np.int32([[[int(0.4*xdim),int(1.0*ydim)],
                                   [int(0.4*xdim),int(0.0*ydim)],
                                   [int(0.65*xdim),int(0.0*ydim)],
                                   [int(0.65*xdim),int(1.0*ydim)]]])

             result = region_of_interest(g_img,roi_mask)

             result = np.dstack(np.array((result,result,result))*255)

             current_img = np.hstack((warped_rgb,result))
             axs[idx].imshow(current_img)
```


0
200
400
600
0
500
1000
1500
2000
2500

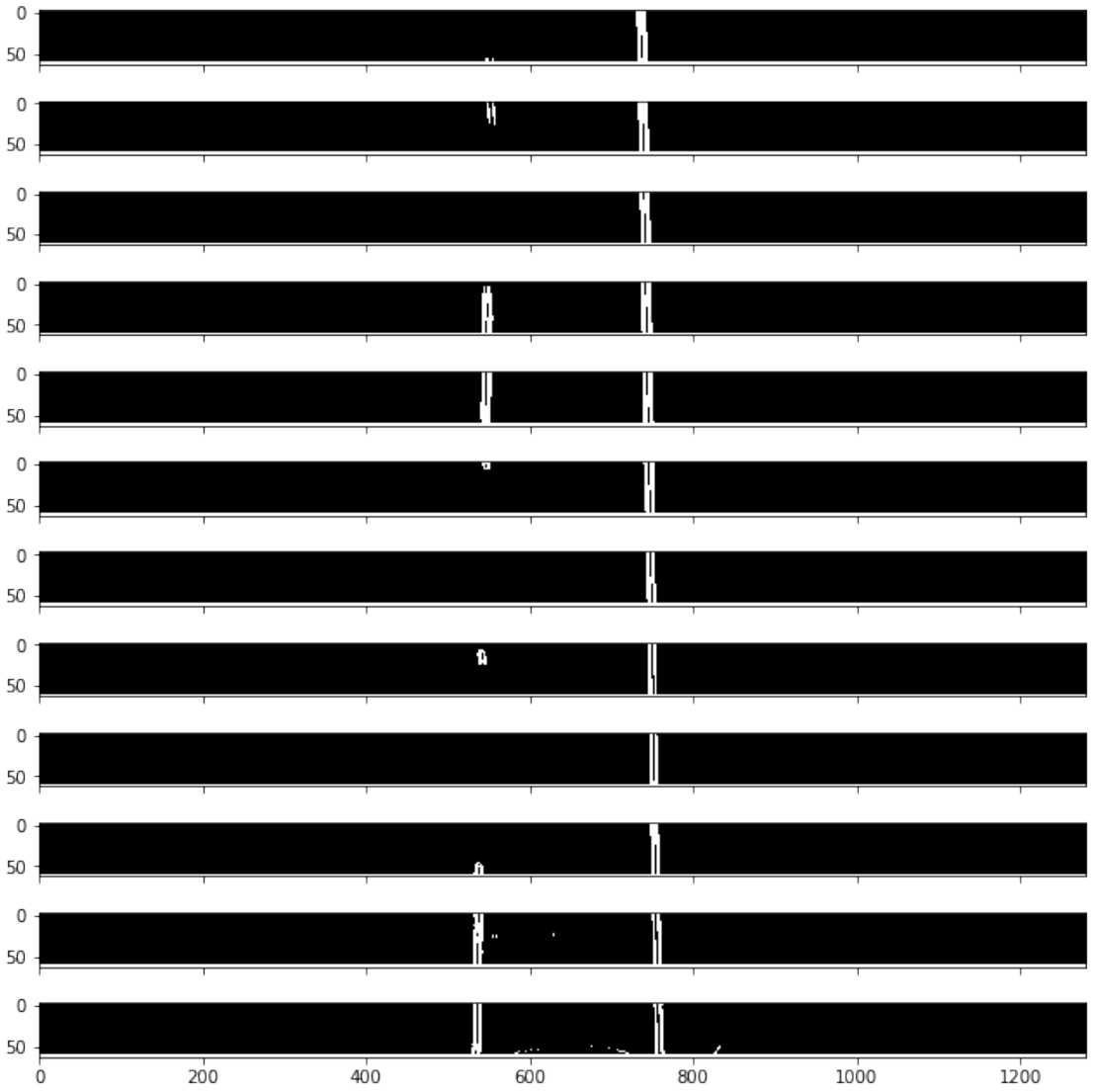
Looks pretty good, though there is some of the warped car in the lane next to the current lane. Perhaps if we were able to ID that segment of pixels was occupied by a car we could mask it from the image in a future task.

1.0.7 Pixel Segregation

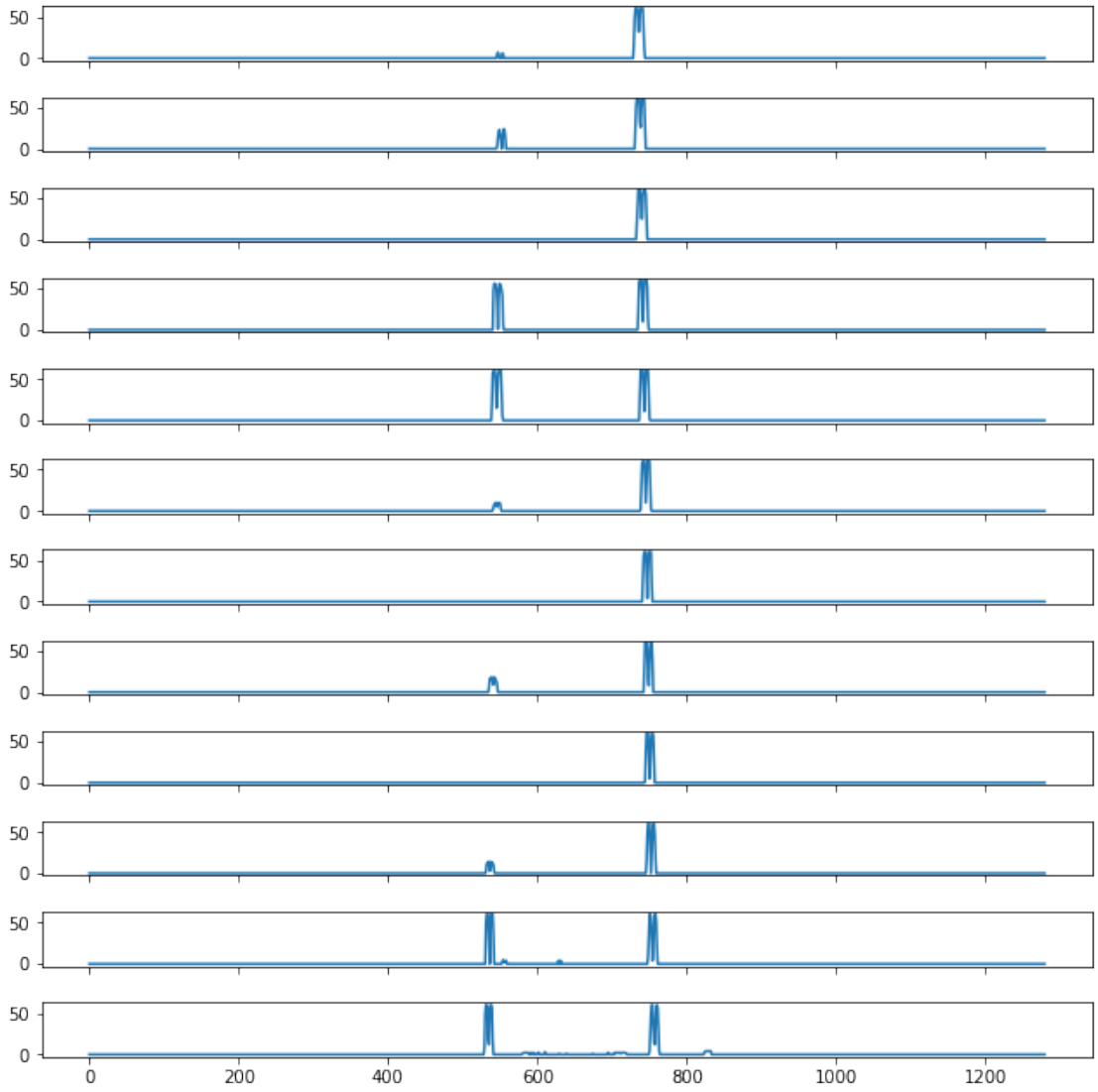
Now we need to fit the BEV lane pixels to 2nd degree polynomials in order to capture the curvature of the lanes. To do that we need to perform a pass over the data and segregate the pixels into left and right lane pixels. An easy way to accomplish this task is to divide the image into about 10 slices along the vertical of the image. For each of the horizontal slices of the image select pixels within a small region about your estimated lane position. Use the selected pixels to estimate the next slice's lane position and repeat the process. This of course needs an initialization step which is done by performing a marginalization of the pixels across the columns (width) of the image and the region of high pixel count on the left half of the image is taken as the seed position of the left lane and the region of high pixel count on the right half of the image is taken as the seed position of the right lane. Some heuristics are employed below to handle cases where there is a low number of pixels in a particular window of a slice as well as the case where no lane pixels are found in any of the slices. Below I demonstrate the slicing of the BEV image and the marginalization of pixel counts along the horizontal to motivate the method.

In [40]:
```
# loading a sample image and applying our pipeline so far
current_img = mpimg.imread(road_image_paths[1])
warped_rgb,inverse_warp_mtx = warp_to_BEV(current_img,camera_mtx, distortion_params)
g_img = hilight_and_edge(warped_rgb,alpha=80.)

# building the mask
ydim,xdim,_ = current_img.shape
roi_mask = np.int32([[[int(0.4*xdim),int(1.0*ydim)],
                      [int(0.4*xdim),int(0.0*ydim)],
                      [int(0.65*xdim),int(0.0*ydim)],
                      [int(0.65*xdim),int(1.0*ydim)]]])

lane_pxls = region_of_interest(g_img,roi_mask)
```

In [41]:
```
# Choose the number slices along the verticle
nwindows = 12

# Set height of windows - based on nwindows above and image shape
window_height = np.int(lane_pxls.shape[0]//nwindows)

# lets look at the slices
f, axs = plt.subplots(nwindows,1, figsize=(9, 9),sharex=True,sharey=True)
f.tight_layout()

for w in range(nwindows):
    slice = lane_pxls[(nwindows-w-1)*window_height:(nwindows-w)*window_height,:]
    axs[-(w+1)].imshow(slice,cmap='gray');
```



In [42]:
```
# we marginalize the pixel count across the verticle per slice
# the mean pixel locations within a particular window selects
# the pixels associated with the lane
f, axs = plt.subplots(nwindows,1, figsize=(9, 9),sharex=True,sharey=False)
f.tight_layout()

for w in range(nwindows):
    slice = lane_pxls[(nwindows-w-1)*window_height:(nwindows-w)*window_height,:]
    histo = np.sum(slice,axis=0)
    axs[-(w+1)].plot(histo);
```



Below I generate a function which finds and segregates the pixels for each lane. The second function produces an image which highlights which pixels are found associated with the left and right lanes.

```
In [43]: def seperate_lane_pxls(bin_img,nwindows=10,window_width=40,pxl_thresh=20):
             # compute window height from the number of windows given
             ydim,xdim = bin_img.shape
             window_height = np.int(ydim/nwindows)

             # get the indices of all nonzero pixels in the original bin image
             hot_pxls = np.vstack(bin_img.nonzero()).T

             # numpy containers to hold pixel coordinates
             left_pxls = np.array([[0,0]])
```

```
right_pxls = np.array([[0,0]])

# computing the initial guess for the lower porition
# of the image lane locations
midline = xdim//2
histoL = np.sum(bin_img[ydim//2:,:midline],axis=0)
histoR = np.sum(bin_img[ydim//2:,midline:],axis=0)
x_l,x_r = np.argmax(histoL),midline+np.argmax(histoR)

# used for computing the mean location of the next slice
# best lane position guess
x_w = np.array([n for n in range(2*window_width)])

for w in range(nwindows):
    ylo,yhi = (nwindows-w-1)*window_height,(nwindows-w)*window_height
    # getting the current rows slice
    img_slice = bin_img[ylo:yhi,:]

    # setting the pixel search window
    xl_lo,xl_hi = np.int32((x_l-window_width,x_l+window_width))
    xr_lo,xr_hi = np.int32((x_r-window_width,x_r+window_width))

    if xl_lo<0:
        xl_lo=0
    if xl_hi>xdim:
        xl_hi=(xdim-1)
    if xr_lo<0:
        xr_lo=0
    if xr_hi>xdim:
        xr_hi=(xdim-1)

    # pixel counts marginalized over the cols
    histL = np.sum(img_slice[:,xl_lo:xl_hi],axis=0)
    histR = np.sum(img_slice[:,xr_lo:xr_hi],axis=0)

    # if the max coutns exceed a threshold than
    # allow window search center updating by
    # average position of pixels in image
    if np.max(histL) > pxl_thresh:
        x_l = x_l-window_width + np.sum(histL*x_w)/np.sum(histL)
    if np.max(histR) > pxl_thresh:
        x_r = x_r-window_width + np.sum(histR*x_w)/np.sum(histR)

    # getting the pixel coords that are in the search window
    x_pxls = hot_pxls[(hot_pxls[:,0] >= ylo) & (hot_pxls[:,0] < yhi) &\
                      (hot_pxls[:,1] >= xl_lo) & (hot_pxls[:,1] < xl_hi)]

    if len(x_pxls) > 0:
```

```
                left_pxls = np.vstack((left_pxls,x_pxls))

            x_pxls = hot_pxls[(hot_pxls[:,0] >= ylo) & (hot_pxls[:,0] < yhi) &\
                              (hot_pxls[:,1] >= xr_lo) & (hot_pxls[:,1] < xr_hi)]

            if len(x_pxls) > 0:
                right_pxls = np.vstack((right_pxls,x_pxls))

        # remove the initialization point from the final results.
        return((left_pxls[1:],right_pxls[1:]))
```

```
In [44]: # function to colorize the left and right lane pixels in the image
         def colorize_lane_pxls(BEV_img):
             (left_pxls,right_pxls) = seperate_lane_pxls(BEV_img,nwindows=10,window_width=40,pxl

             hilight_right = np.zeros_like(BEV_img)

             for pxly,pxlx in right_pxls:
                 hilight_right[pxly,pxlx] = 255

             hilight_left = np.zeros_like(BEV_img)

             for pxly,pxlx in left_pxls:
                 hilight_left[pxly,pxlx] = 255

             combined = np.dstack([hilight_right,hilight_left,BEV_img*255])

             return combined
```

Let's see how well the selected parameters work on being able to detect and select left vs right lane pixels in the test images. Below the left lane is colored cyan while the right lane is colored magenta, blue pixels do not belong to either lane and are not used for computing any of the properties of the lanes.

```
In [45]: # apply the lane segregation technique to each test image
         f, axs = plt.subplots(len(road_image_paths),1,figsize=(20,20))
         f.tight_layout()

         for idx in range(len(road_image_paths)):
             current_img = mpimg.imread(road_image_paths[idx])
             warped_rgb,inverse_warp_mtx = warp_to_BEV(current_img,camera_mtx, distortion_params
             g_img = hilight_and_edge(warped_rgb,alpha=80.)

             # building the mask
             ydim,xdim,_ = current_img.shape
             roi_mask = np.int32([[[int(0.4*xdim),int(1.0*ydim)],
                                   [int(0.4*xdim),int(0.0*ydim)],
                                   [int(0.65*xdim),int(0.0*ydim)],
```

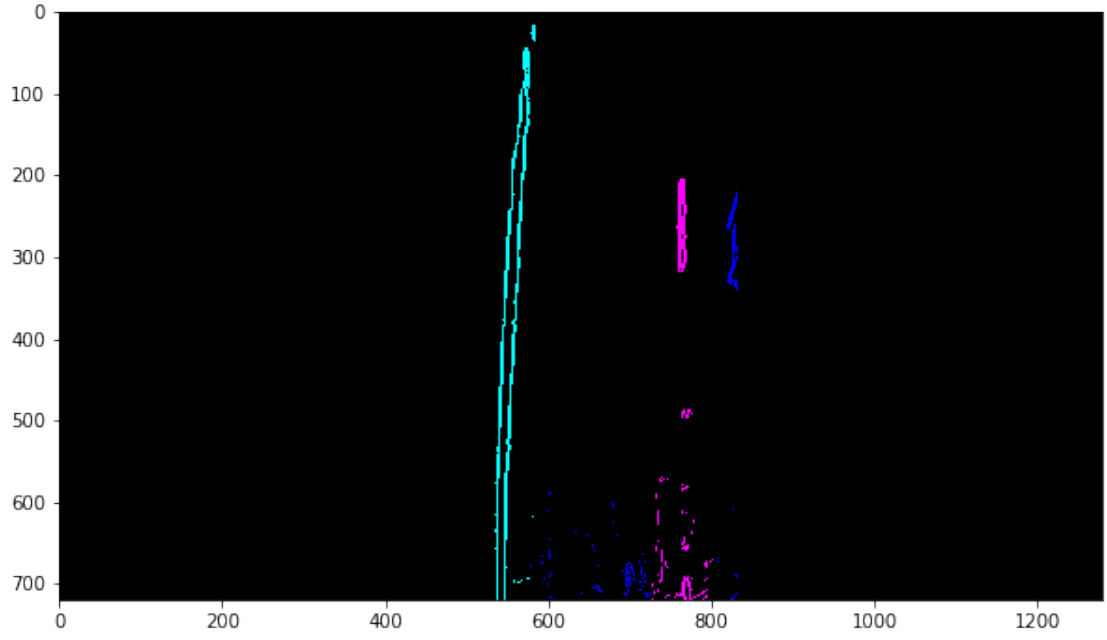
```
                    [int(0.65*xdim),int(1.0*ydim)]]])

lane_pxls = region_of_interest(g_img,roi_mask)
colorized_warp = colorize_lane_pxls(lane_pxls)

current_img = np.hstack((current_img,colorized_warp))
axs[idx].imshow(current_img)
```


0
200
400
600
0
500
1000
1500
2000
2500

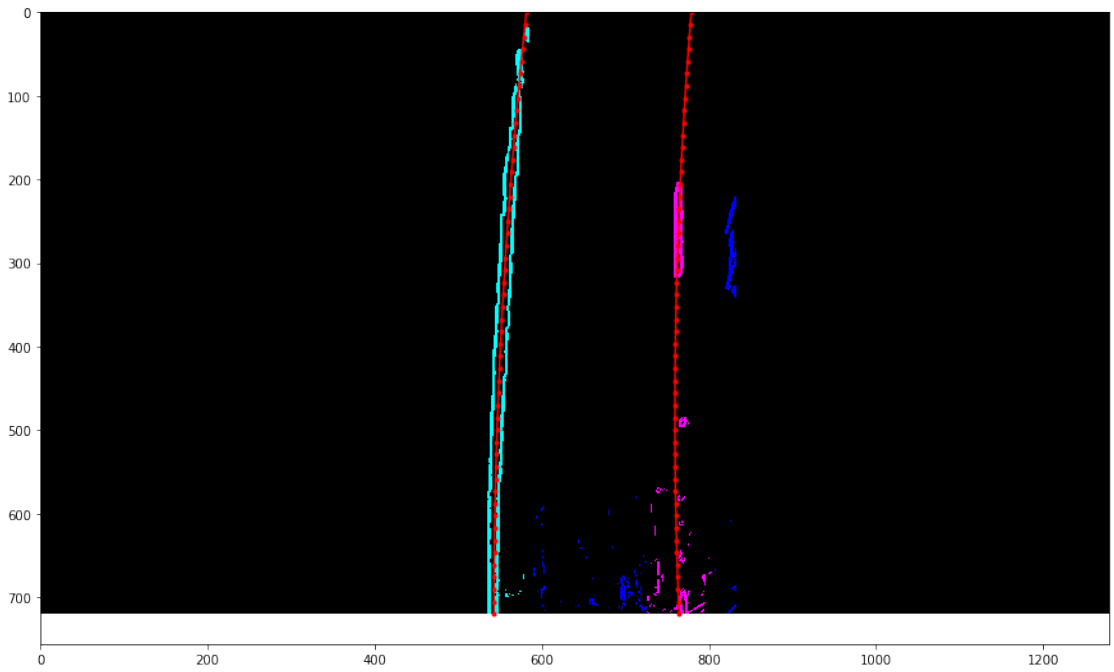
1.0.8 Lane Radius of Curvature and the Position of the Vehicle

To compute the radius of curvature (ROC) of the road and the position of the camera with respect to the lane center we need to fit a 2nd order polynomial to each lane set of pixels. Once the fit parameters are found, computing the radius of curvature is a simple exercise in calculus and scaling from pixel measurements to physical distances, using a known curve ROC as a calibration step. Below, I simply fit two 2nd order polynomials to each set of lane pixels and plot the results as an example of the types of fit we are looking for.

In [46]:
```
# loading our example image and applying the pipeline so far
current_img = mpimg.imread(road_image_paths[2])
warped_rgb,inverse_warp_mtx = warp_to_BEV(current_img,camera_mtx, distortion_params)
g_img = hilight_and_edge(warped_rgb,alpha=80.)

# building the mask
ydim,xdim,_ = current_img.shape
roi_mask = np.int32([[[int(0.4*xdim),int(1.0*ydim)],
                      [int(0.4*xdim),int(0.0*ydim)],
                      [int(0.65*xdim),int(0.0*ydim)],
                      [int(0.65*xdim),int(1.0*ydim)]]])

lane_pxls = region_of_interest(g_img,roi_mask)
(left_pxls,right_pxls) = seperate_lane_pxls(lane_pxls)
colorized_warp = colorize_lane_pxls(lane_pxls)

plt.figure(figsize=(10,7))
plt.imshow(colorized_warp);
```



```
In [47]: # fit the segergated pixel points to 2nd order polys which are in are (y,x) format
         lft_params = np.polyfit(left_pxls[:,0],left_pxls[:,1],2)
         rgt_params = np.polyfit(right_pxls[:,0],right_pxls[:,1],2)

In [48]: # computing the points to plot on the image after the fit
         sample_y = np.linspace(0,lane_pxls.shape[0])

         poly_left = lft_params[0]*sample_y**2 + lft_params[1]*sample_y + lft_params[2]
         poly_right = rgt_params[0]*sample_y**2 + rgt_params[1]*sample_y + rgt_params[2]

In [49]: # showing the fits against the lane pixels
         plt.figure(figsize=(15,10))
         plt.imshow(colorized_warp);
         plt.plot(poly_left,sample_y,color='r',marker='.');
         plt.plot(poly_right,sample_y,color='r',marker='.');
```



When we have good pixel detection for each lane the fits are nearly perfect. With these fits we can now estimate the curvature and deduce where along the horizontal the center of the lane lies at the position of the camera. When we know where the center of the lane is by computing the difference between the mid point of the image and lane center position we can know how far from center the camera is in any frame.

```
In [50]: # Recompute with pixel to m scaling which was found from
         # calibration against a known ROC turn data
```

```
        ym_per_pix = (3.5*3)/720 # meters per pixel in y dimension
        xm_per_pix = 3.7/794 # meters per pixel in x dimension

        lft_params = np.polyfit(ym_per_pix*left_pxls[:,0],xm_per_pix*left_pxls[:,1],2)
        rgt_params = np.polyfit(ym_per_pix*right_pxls[:,0],xm_per_pix*right_pxls[:,1],2)

        # Computing Center Deviation at car center
        eval_y = warped_rgb.shape[0]*ym_per_pix
        poly_left = lft_params[0]*eval_y**2 + lft_params[1]*eval_y + lft_params[2]
        poly_right = rgt_params[0]*eval_y**2 + rgt_params[1]*eval_y + rgt_params[2]

        car_lane_pos = (poly_left+poly_right)/2

        print("center of the lane: {} (pixels)".format(car_lane_pos/xm_per_pix))

        mid_pt = warped_rgb.shape[1]/2.*xm_per_pix

        deviation = mid_pt-car_lane_pos

        # negative indicates we're left of center, positive right of center
        print("deviation from center: {} (cm)".format(deviation*100))
```

```
center of the lane: 653.4893408767416 (pixels)
deviation from center: -6.285964892184381 (cm)
```

```
In [51]: # Radius of Curvature
         left_ROC = (1.+(2*lft_params[0]*eval_y + lft_params[1])**2)**1.5/np.abs(2*lft_params[0]
         right_ROC = (1.+(2*rgt_params[0]*eval_y + rgt_params[1])**2)**1.5/np.abs(2*rgt_params[0

         # compute the radius of curvature as a weighted average
         # between the two lane ROC values. The weight is determined
         # by the number of pixels in each lane and thus confidence in
         # an accurate estimate of the true lane curve.
         nleftpxl = len(left_pxls)
         nrightpxl = len(right_pxls)
         ROC = (nleftpxl*left_ROC+nrightpxl*right_ROC)/(nleftpxl+nrightpxl)

         print("Radius of curvature: {} (m)".format(ROC))
```

```
Radius of curvature: 298.760406780486 (m)
```

Below I wrap up this entire process into a function which also returns multiple points on the unwarped image which maps out a polygon of the identified lane area. I will note that at this point the function is not robust against cases where no lane pixels are found or one lane has found pixels and none for the other. That case is handled later.

```
In [52]: def camera_deviation_ROC(rgb_img,left_pxls,right_pxls):
             ydim,xdim,_ = rgb_img.shape
```

```
ym_per_pix = (3.5*3)/ydim # meters per pixel in y dimension
xm_per_pix = 3.7/xdim # meters per pixel in x dimension

# fitting the lane pixels
lft_params = np.polyfit(ym_per_pix*left_pxls[:,0],xm_per_pix*left_pxls[:,1],2)
rgt_params = np.polyfit(ym_per_pix*right_pxls[:,0],xm_per_pix*right_pxls[:,1],2)

# Computing Center Deviation at car center
eval_y = ydim*ym_per_pix
poly_left = lft_params[0]*eval_y**2 + lft_params[1]*eval_y + lft_params[2]
poly_right = rgt_params[0]*eval_y**2 + rgt_params[1]*eval_y + rgt_params[2]

# computing pixel points to draw a filling
# poly on the BEV image
upper_pxl_y = (0.0*ydim)*ym_per_pix

# computing the points for the polygon that outlines the road way
left_lane_upper_x = [[(lft_params[0]*ypt**2 + lft_params[1]*ypt+ lft_params[2])/xm_
right_lane_upper_x = [[(rgt_params[0]*ypt**2 + rgt_params[1]*ypt + rgt_params[2])/x

left_lane_upper_x = np.int32(left_lane_upper_x)
right_lane_upper_x = np.int32(right_lane_upper_x[::-1])

# the polygon points to draw the lane curvature on the lane
poly_pts = np.vstack((left_lane_upper_x,right_lane_upper_x))

# computing the lane center position
# as measured by the camera
car_lane_pos = (poly_left+poly_right)/2

# computing where the camera is located
# along the horizontal
mid_pt = xdim/2.*xm_per_pix

# whats the difference between the camera
# position and the infered lane center
deviation = mid_pt - car_lane_pos

# Radius of Curvature
left_ROC = (1.+(2*lft_params[0]*eval_y + lft_params[1])**2)**1.5/np.abs(2*lft_param
right_ROC = (1.+(2*rgt_params[0]*eval_y + rgt_params[1])**2)**1.5/np.abs(2*rgt_para

# pixel weighted ROC
nleftpxl = len(left_pxls)
nrightpxl = len(right_pxls)
ROC = (nleftpxl*left_ROC+nrightpxl*right_ROC)/(nleftpxl+nrightpxl)

return((deviation,ROC,poly_pts))
```

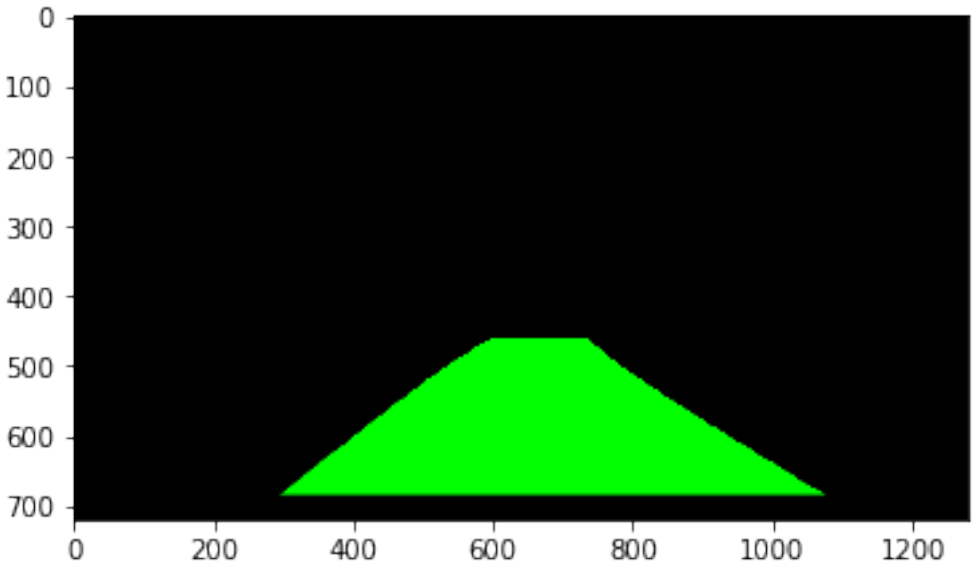
```
In [53]: (deviation,ROC,pxl_poly_pts)=camera_deviation_ROC(warped_rgb,left_pxls,right_pxls)
```

1.0.9 Draw the Detected Lane Region On the Original Image

Given that the previous function returned the unwarped polygon lane region pixel positions, we want to fill a polygon with a particular color and opacity on the original image. Hopefully the polygon that is overlaid on the image should show the appropriate curvature and track the lane we are in correctly.

```
In [54]: def get_mask_projection(rgb_img,pxl_poly_pts,inverse_proj_mtx):
             mask = np.zeros_like(rgb_img[:,:,0])

             # filling pixels inside the polygon defined by "vertices"
             cv2.fillConvexPoly(mask, np.array([pxl_poly_pts]),255)

             # returning the image only where mask pixels are nonzero
             masked = cv2.bitwise_and(rgb_img[:,:,0], mask)

             # reversing the projection
             poly_lanes = cv2.warpPerspective(mask, inverse_proj_mtx, mask.shape[::-1], flags=cv

             # making the mask colored
             proj_rgb = np.dstack((np.zeros_like(poly_lanes),poly_lanes,np.zeros_like(poly_lanes

             return proj_rgb
```

```
In [55]: # the overlay polygon before overlaying on the original image
         proj_rgb = get_mask_projection(warped_rgb,pxl_poly_pts,inverse_warp_mtx)
         plt.imshow(proj_rgb);
```



```
In [56]: # overlaying the lane polygon on the original image with some opacity effect
         overlay = cv2.addWeighted(proj_rgb, 0.3, current_img, 0.7, 0)
         plt.figure(figsize=(15,10))
         plt.imshow(overlay);
```

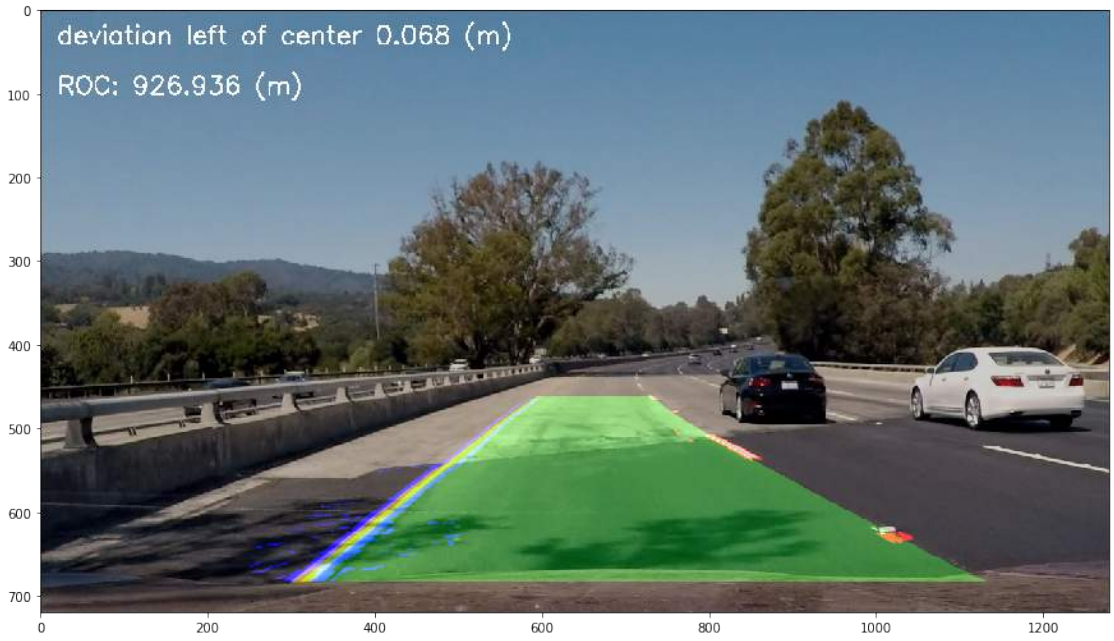
Great! Now all of the steps: undistorting an image, perspective transformation, color transforming and edge detection, lane pixel segregation, modeling the lane curvature, and producing a polygon overlay are wrapped into the single function below. This version of the function is indented to work on a single image in similar lighting and roadway conditions. In addition to overlaying the detected lane polygon, I've also added highlighting the pixels which are detected as the left and right lanes on the original image and I also include the deviation from center and ROC within the resulting images as text in the image.

```
In [57]: def find_lanes_image(RGB_img,camera_mtx,distortion_params):

             # Undistort and then Birds-Eye-View Transform
             warped_RGB,Inverse_warp_mtx = warp_to_BEV(RGB_img, camera_mtx, distortion_params)
             G_img = hilight_and_edge(warped_RGB,alpha=80.)

             # building the mask
             ydim,xdim,_ = RGB_img.shape
             roi_mask = np.int32([[[int(0.4*xdim),int(1.0*ydim)],
                                   [int(0.4*xdim),int(0.0*ydim)],
                                   [int(0.65*xdim),int(0.0*ydim)],
                                   [int(0.65*xdim),int(1.0*ydim)]]])
             # applying the mask
             Lane_pxls = region_of_interest(G_img,roi_mask)
```

```
# segregating the lane pixels
(Left_pxls,Right_pxls) = seperate_lane_pxls(Lane_pxls)

# drawing the lane pixels mask
Hilight_right = np.zeros_like(warped_RGB)

for pxly,pxlx in Right_pxls:
    Hilight_right[pxly,pxlx] = np.array([255,0,0])

Hilight_left = np.zeros_like(warped_RGB)

for pxly,pxlx in Left_pxls:
    Hilight_left[pxly,pxlx] = np.array([0,0,255])

Lane_marks = cv2.addWeighted(Hilight_left, 1.0, Hilight_right, 1.0, 0)
# un warping them
Lane_marks = cv2.warpPerspective(Lane_marks, Inverse_warp_mtx, Lane_marks[:,:,0].sh

# fitting the pixel curves and deducing parameters
(Deviation,ROC,Pxl_poly_pts) = camera_deviation_ROC(warped_RGB,Left_pxls,Right_pxls

# left vs right of center
if Deviation < 0.0:
    Direction = 'left'
else:
    Direction = 'right'

# generating the overlay polygon
Proj_rgb = get_mask_projection(warped_RGB,Pxl_poly_pts,Inverse_warp_mtx)

# merging the images
Overlay = cv2.addWeighted(RGB_img,0.8,Lane_marks,1.0,0.0)
Overlay = cv2.addWeighted(Overlay,1.0,Proj_rgb,0.3,0.0)

# adding the textual information to the images
cv2.putText(Overlay,
            'deviation {} of center {} (m)'.format(Direction,round(abs(Deviation),3
            (20,40),
            cv2.FONT_HERSHEY_SIMPLEX,
            1,
            (255,255,255),
            2)

cv2.putText(Overlay,
            'ROC: {} (m)'.format(round(ROC,3)),
            (20,100),
            cv2.FONT_HERSHEY_SIMPLEX,
```

```
                    1,
                    (255,255,255),
                    2)


        return Overlay
```

In [58]:
```
# example application of warper function
img = mpimg.imread(road_image_paths[0])
overlay = find_lanes_image(img, camera_mtx, distortion_params)

plt.figure(figsize=(15,10))
plt.imshow(overlay);
```



In [59]:
```
# testing the parameters over all the test images
f, axs = plt.subplots(len(road_image_paths),1,figsize=(10,30))
f.tight_layout()

for idx in range(len(road_image_paths)):
    current_img = mpimg.imread(road_image_paths[idx])
    overlay = find_lanes_image(current_img,camera_mtx,distortion_params)

    axs[idx].imshow(overlay)
```


deviation left of center 0.093 (m)
ROC: 926.836 (m)
deviation left of center 0.015 (m)
ROC: 10215.855 (m)
deviation left of center 0.039 (m)
ROC: 481.612 (m)
deviation left of center 0.017 (m)
ROC: 1023.523 (m)
deviation left of center 0.038 (m)
ROC: 862.786 (m)
deviation left of center 0.053 (m)
ROC: 268.425 (m)
deviation left of center 0.056 (m)
ROC: 485.502 (m)
deviation left of center 0.004 (m)
ROC: 2715.864 (m)

1.0.10 Applying Lane Tracking to Video

Now we apply the pipeline to a video stream. For this case we make a few modifications, such as exponential smoothing of the ROC, lane deviation and smoothing to the lane estimation parameters. This is to ensure that no single (potentially noisy) frame unduly influences the results of the lane tracking. I also employee heuristics to handle the edge cases where no lane pixels are detected for the left or right or both lanes. In cases where only one lane is detected I simply translate the estimated parameters over to the usual location of the other lane. In cases where there are no lane pixels detected I default to straight lane parameters, which hopefully is one for at most 1 frame of the video.

```
In [60]: def lane_Dev_ROC_Priors(rgb_img,left_pxls,right_pxls,prior_l,prior_r,prior_dev,prior_rc
             ydim,xdim,_ = rgb_img.shape
             ym_per_pix = (3.5*3)/ydim # meters per pixel in y dimension
             xm_per_pix = 3.7/xdim # meters per pixel in x dimension

             # defaults when pixels arent findable
             if len(left_pxls) != 0:
                 lft_params = np.polyfit(ym_per_pix*left_pxls[:,0],xm_per_pix*left_pxls[:,1],2)
             else:
                 lft_params = np.array([0.0,0.0,xm_per_pix*550])

             if len(right_pxls) != 0:
                 rgt_params = np.polyfit(ym_per_pix*right_pxls[:,0],xm_per_pix*right_pxls[:,1],2
             else:
                 rgt_params = np.array([0.0,0.0,xm_per_pix*785])

             if len(left_pxls) != 0 and len(right_pxls) == 0:
                 rgt_params = lft_params
                 rgt_params[-1] = xm_per_pix*785

             if len(left_pxls) == 0 and len(right_pxls) != 0:
                 lft_params = right_pxls
                 lft_params[-1] = xm_per_pix*785

             # Exponential Decay Factor
             gamma = 0.2

             if prior_l != None:
                 lft_params = (1.-gamma)*prior_l + gamma*lft_params
             if prior_r != None:
                 rgt_params = (1.-gamma)*prior_r + gamma*rgt_params

             # Computing Center Deviation at car center
             eval_y = ydim*ym_per_pix
```

```
        poly_left = lft_params[0]*eval_y**2 + lft_params[1]*eval_y + lft_params[2]
        poly_right = rgt_params[0]*eval_y**2 + rgt_params[1]*eval_y + rgt_params[2]

        # computing pixel points to draw a filling
        # poly on the BEV image
        upper_pxl_y = (0.0*ydim)*ym_per_pix

        left_lane_upper_x = [[(lft_params[0]*ypt**2 + lft_params[1]*ypt+ lft_params[2])/xm_
        right_lane_upper_x = [[(rgt_params[0]*ypt**2 + rgt_params[1]*ypt + rgt_params[2])/x

        left_lane_upper_x = np.int32(left_lane_upper_x)
        right_lane_upper_x = np.int32(right_lane_upper_x[::-1])

        poly_pts = np.vstack((left_lane_upper_x,right_lane_upper_x))

        car_lane_pos = (poly_left+poly_right)/2

        mid_pt = xdim/2.*xm_per_pix

        deviation = mid_pt - car_lane_pos

        # expoential smoothing
        if prior_roc != None:
            deviation = (1.-gamma)*prior_dev + gamma*deviation

        # Radius of Curvature
        left_ROC = (1.+(2*lft_params[0]*eval_y + lft_params[1])**2)**1.5/np.abs(2*lft_param
        right_ROC = (1.+(2*rgt_params[0]*eval_y + rgt_params[1])**2)**1.5/np.abs(2*rgt_para

        # pixel weighted ROC
        nleftpxl = len(left_pxls)
        nrightpxl = len(right_pxls)
        ROC = (nleftpxl*left_ROC+nrightpxl*right_ROC)/(nleftpxl+nrightpxl)

        # exponential smoothing
        if prior_roc != None:
            ROC = (1.-gamma)*prior_roc + gamma*ROC

        # returing the usual parameters + values to use for exponential smoothing
        return((deviation,ROC,poly_pts,lft_params,rgt_params,deviation,ROC))
```

In [61]:
```
def find_lanes_video(RGB_img,camera_mtx,distortion_params):

        # Undistort and then Birds-Eye-View Transform
        warped_RGB,Inverse_warp_mtx = warp_to_BEV(RGB_img, camera_mtx, distortion_params)
        G_img = hilight_and_edge(warped_RGB,alpha=80.)

        # building the mask
```

```
ydim,xdim,_ = RGB_img.shape
roi_mask = np.int32([[[int(0.4*xdim),int(1.0*ydim)],
                      [int(0.4*xdim),int(0.0*ydim)],
                      [int(0.65*xdim),int(0.0*ydim)],
                      [int(0.65*xdim),int(1.0*ydim)]]])
# applying the mask
Lane_pxls = region_of_interest(G_img,roi_mask)

# segregating the lane pixels
(Left_pxls,Right_pxls) = seperate_lane_pxls(Lane_pxls)

# drawing the lane pixels mask
Hilight_right = np.zeros_like(warped_RGB)

for pxly,pxlx in Right_pxls:
    Hilight_right[pxly,pxlx] = np.array([255,0,0])

Hilight_left = np.zeros_like(warped_RGB)

for pxly,pxlx in Left_pxls:
    Hilight_left[pxly,pxlx] = np.array([0,0,255])

if (len(Left_pxls) != 0) and (len(Right_pxls) != 0):
    Lane_marks = cv2.addWeighted(Hilight_left, 1.0, Hilight_right, 1.0, 0)
    # un-warping them
    Lane_marks = cv2.warpPerspective(Lane_marks, Inverse_warp_mtx, Lane_marks[:,:,0

# global parameters to handle the multiframe smoothing
global prior_l
global prior_r
global prior_dev
global prior_roc
# fitting the pixel curves and deducing parameters
(Deviation,ROC,Pxl_poly_pts,prior_l,prior_r,prior_dev,prior_roc) = lane_Dev_ROC_Pri
                                                                   Left_pxls,
                                                                   Right_pxls,
                                                                   prior_l,
                                                                   prior_r,
                                                                   prior_dev,
                                                                   prior_roc)


# left vs right of center
if Deviation < 0.0:
    Direction = 'left'
else:
    Direction = 'right'

# generating the overlay polygon
```

```
        Proj_rgb = get_mask_projection(warped_RGB,Pxl_poly_pts,Inverse_warp_mtx)

        # merging the images
        Overlay = cv2.addWeighted(RGB_img,0.9,Lane_marks,1.0,0.0)
        Overlay = cv2.addWeighted(Overlay,1.0,Proj_rgb,0.3,0.0)

        # overlay the textual information
        cv2.putText(Overlay,
                    'deviation {} of center {} (m)'.format(Direction,round(abs(Deviation),3
                    (20,40),
                    cv2.FONT_HERSHEY_SIMPLEX,
                    1,
                    (255,255,255),
                    2)

        cv2.putText(Overlay,
                    'ROC: {} (m)'.format(round(ROC,3)),
                    (20,100),
                    cv2.FONT_HERSHEY_SIMPLEX,
                    1,
                    (255,255,255),
                    2)


        return Overlay
```

In [62]:
```
# Import everything needed to edit/save/watch video clips
from moviepy.editor import VideoFileClip
from IPython.display import HTML
```

In [63]:
```
# store the results in the file name below
annotated_output1 = 'project_video_annotated_smoothed.mp4'
clip1 = VideoFileClip('project_video.mp4')
```

In [64]:
```
def process_image(img):
    return find_lanes_video(img,camera_mtx,distortion_params)
```

In [65]:
```
# intialize the exponential smoothing
# values
prior_l = None
prior_r = None
prior_dev = None
prior_roc = None

# Uncomment the following lines to generate the annotated
# video feed
#annotated_clip1 = clip1.fl_image(process_image)
#%time annotated_clip1.write_videofile(annotated_output1, audio=False)
```

I provide a link to the results of the processing here annotated video results (project_video_annotated_smoothed.mp4)

1.0.11 Where to Improve

Performance on this pipeline can be improved by trying to reduce the lane noise, and by using prior frame information more advantageously. In the Udacity course content they outline a method where rather than using the slices and windows to select the lane pixels most likely associated with each lane, one would only perform that action once, and there after use the previously estimated lane curve to find the slight modification to the pixels for the next frame. This could give a speed up to the estimation method but was not employed here. A more robust method would be to model the lane locations for each slice as a Bayesian belief network, where the previous slice would act as a prior belief of the lane locations for the next slice. This would allow much sharper road curvature and allow a quantitative method for estimating confidence in the methods ability to track the lane. This could allow the system to indicate to the driver when it was not confident in its lane tracking and to take control at that point. It should also go without saying that I need to fail elegantly, ie when no lanes can be found I need to make sure that this pipeline is robust against any run-time errors as well.

Further improvements could be made by being able to identify which parts of the original images have other vehicles in frame so that those pixels could be masked for the lane detection process. In some of the frames you can see that the car next to the lane we are tracking gets identified as likely part of the right lane edge pixels. Being able to mask that part of the image would reduce the pixel noise. It should also be possible to select edges where the area between nearby and having nearly the same orientation edges should have an acceptable hue towards yellow lane markers, or some acceptable high levels in the RGB channels to track white. This would require that the edges you find are part of a thick lane marker rather than a horizontal change in the lane texture/type. You can see in the linked video how poorly the pipeline performs when there is this horizontal lane textuure/type running down the outer to middle of the lane here (challenge_video_annotated.mp4).

We could also improve the lane tracking by including mapping data on top of the computer vision technique. If we had good localization of the camera, along with an up-to-date map of the road we are on we could use that information as a prior to the lane parameters and generate more robust estimation that would need fewer lane pixels detected (even over multiple frames) to be able to converge on the location of the lanes.

Another area we could improve is by having a Bayesian prior on the potential shape of the unwarped polygon such that extereme changes in the shape should be really unlikely.

One major improvement I believe deserves lots of attention is an adaptive warping point system. As currently employed the points in the image which are warped are fixed, but when you have changing inclination of the road, or sharp turns these warping points do not accurately reflect the birds-eye-view image of the road. I am unsure how to algorithmically detect where the warping points should be placed but I believe this would offer a lot of improvement if one could solve that problem. Finally, I believe that dynamic contrasting and brightness controls across the image could improve the method as in the test case we do not see a large variety in lighting conditions, but I'm not sure how well this method would work in different lighting conditions.

Below is a short clip of another video which even over a short segment shows many further challenges which this current method does not solve, such as motorcycles over taking you, sharp turns, and rapidly changing lighting conditions. Take a look at this link to see how the current pipeline performs in these conditions pipeline on a challenging road

(harder_challenge_video_smoothed.mp4).

```
In [67]: # generating the challenge video where there is large
         # texture change running along the horizontal along the middle
         # of the lane
         prior_l = None
         prior_r = None
         prior_dev = None
         prior_roc = None

         # uncomment to generate full video
         #annotated_output2 = 'challenge_video_smoothed.mp4'
         #clip2 = VideoFileClip('challenge_video.mp4')
         #annotated_clip2 = clip2.fl_image(process_image)
         #%time annotated_clip2.write_videofile(annotated_output2, audio=False)
```

```
[MoviePy] >>>> Building video challenge_video_smoothed.mp4
[MoviePy] Writing video challenge_video_smoothed.mp4


100%|| 485/485 [04:23<00:00,  1.88it/s]


[MoviePy] Done.
[MoviePy] >>>> Video ready: challenge_video_smoothed.mp4

CPU times: user 3min 36s, sys: 644 ms, total: 3min 37s
Wall time: 4min 26s
```

```
In [ ]: # generation of the challenging video is done here.
        from moviepy.editor import *
        from moviepy.video.io.ffmpeg_tools import ffmpeg_extract_subclip

        annotated_output3 = 'harder_challenge_video_smoothed.mp4'
        subclip = VideoFileClip('harder_challenge_video.mp4').subclip(0,15)
```

```
In [ ]: prior_l = None
        prior_r = None
        prior_dev = None
        prior_roc = None

        # uncomment to generate video
        #annotated_clip3 = subclip.fl_image(process_image)
        #%time annotated_clip3.write_videofile(annotated_output3, audio=False)
```