Advanced Lane Finding Project

The goals / steps of this project are :

- 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.

[Rubric] (https://review.udacity.com/#!/rubrics/571/view (https://review.udacity.com/#!/rubrics/571/view)) Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

---

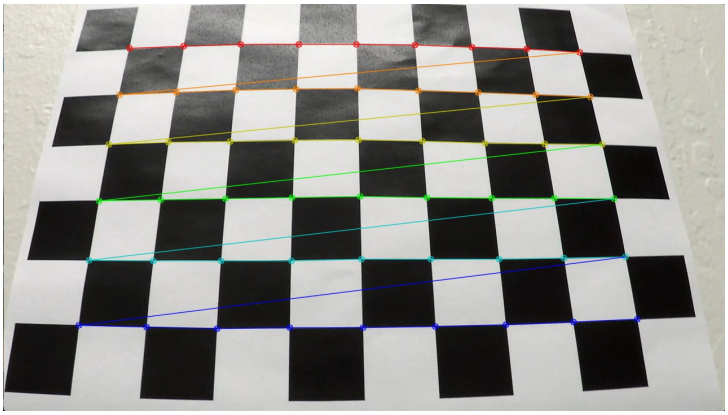
1. Camera Calibration

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

The code for this step is contained in the code cells 3 and 4 of the IPython notebook located in "./Advanced Lane Lines-v02.ipynb"

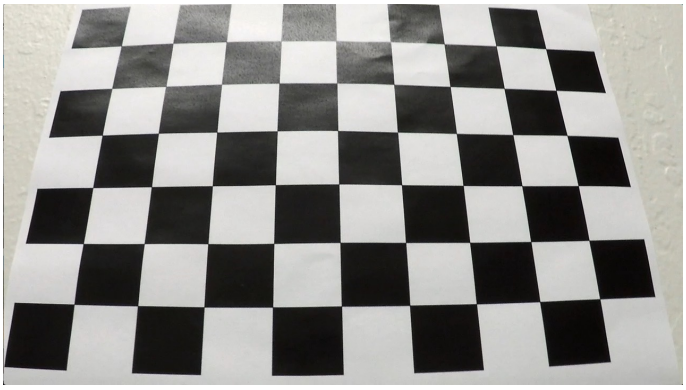
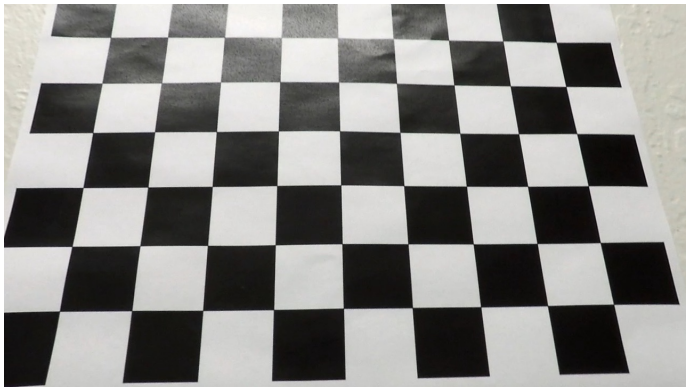
Camera lens usually result in image distortions with edges curved outside or inside. To remove these distortions, I used chessboard images and OpenCV functions findChessboardCorners, drawChessboardCorners, calibrateCamera and undistort functions. I used UDacity provide 20 chessboard images for calibration. Sample chessboard images are read but only one image is shown in code file and in this document below. This image has 9 inner corners in x-direction and 6 inner corners in the y-direction. findChessboardCorners function identifies these 9x6 corner points and we can draw these 9x6 corner points using drawChessboardCorners function for visual display.

Chessboard corners



For calibration, calibrateCamera function uses two arrays (objPoints and imgpoints). objpoints are initial 9x6 corners and imgpoints are the 9x6 corners which should be there in an undistorted image. Then calibrateCamera function is used to calibrate these points to remove the edge distortions because of camera lens. This function returns the matrices mtx and dist which are used to undistort the images. I saved these matrices in the pickle file which can then be used later for removing distortions.

Using the above mtx and dist matrices, undistort function is called to remove distortions. Distorted and Undistorted images are shown below

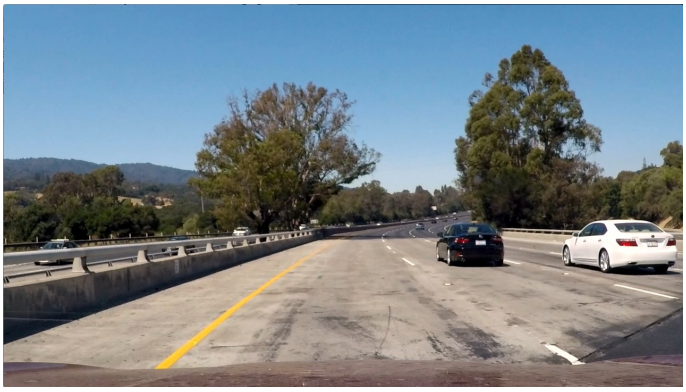
Distorted Image - calibration3.jpg	**Undistorted Image - calibration3_undist.jpg**
	

Pipeline (single images)

1. Provide an example of a distortion-corrected image.

I loaded the mtx and dist matrices from pickle dump saved above. To undistort a test image, a test image is selected and Openv undistort function is applied to this test image using mtx and dist matrices.

The code for this step is contained in the code cell 5 of the IPython notebook located in "./Advanced Lane Lines-v02.ipynb"

Distorted Image - test1.jpg	Undistorted Image - undist_test1.jpg
	

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

I used a combination of gradient and color thresholds to generate a binary imaage. I used opencv sobel function for x and y gradients and used S, L and V channels for color segregation.

function abs_sobel_thresh is defined for x and y gradients

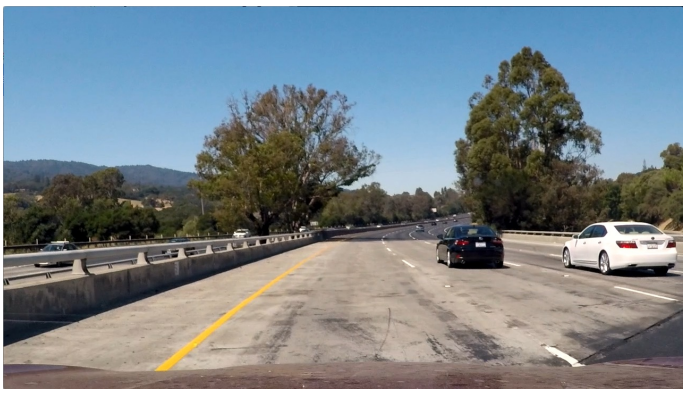
function color_threshold is defined for color gradients

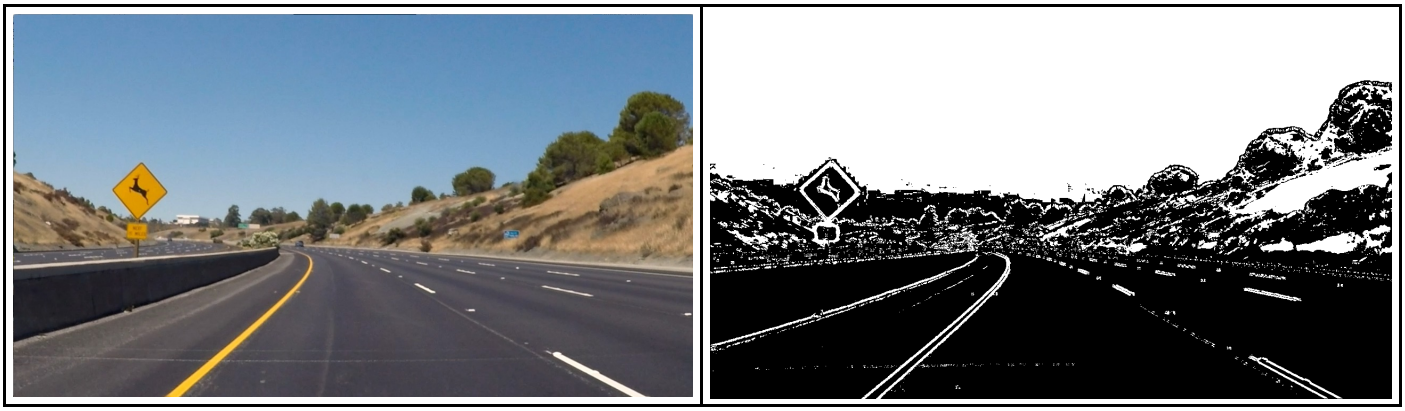
These two functions are then combined in createBinary funtion.

The code for this step is contained in the code cells 6 to 12 of the IPython notebook located in "./Advanced Lane Lines-v02.ipynb"

- x grad thresholds are (12, 255)
- y grad thresholds are (25, 255)
- S channel thresholds are (80, 255)
- L channel thresholds are (50, 200)
- V channel thresholds are (50, 225)

6 original and their binary images are shown below

Original Image	Binary Image
	



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

The code for creating perspective transform is contained in cell 13 of the IPython notebook located in "./Advanced Lane Lines-v02.ipynb" .

Function "warper" is defined to create perspective transform. OpenCv getPerspectiveTransform is used to generate transform matrix and then warpPerspective function is used to generate perspective transform.

getPerspectiveTransform is used to generate reverse perspective transform also.

For generating transform matrix, src and dest image points are defined as below

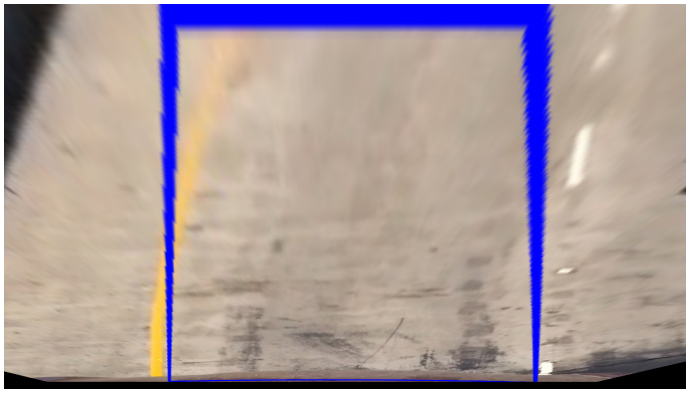
src = np.float32([[575, 460], [300, 705], [1115, 705], [700, 460]])

dst = np.float32([[310, 0], [310, 700], [1000,700], [1000, 0]])

I chose the hardcode the source and destination points in the following manner:

Source	Destination
575, 460	310, 0
300, 705	310, 700
1115, 705	1000, 700
700, 460	1000, 0

I verified that my perspective transform was working as expected by drawing the `src` and `dst` points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image.

Original Image with src points	Warped Image
	

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

To identify left and right lanes, histogram is generated by summing pixel values for the lower half of the image along y -axis. The peaks in the image show the bottom point of the left and right lanes.

Then a window of width 80 is defined centered around left lane bottom point and same is done for the right lane.

Total 9 windows are defined in sequence from bottom to top for each lane.

After identifying window, pixels are identified in each window and their middle value is calculated to find the middle point of the window above the bottom window. This procedure is repeated for 9 windows until top of the image is reached.

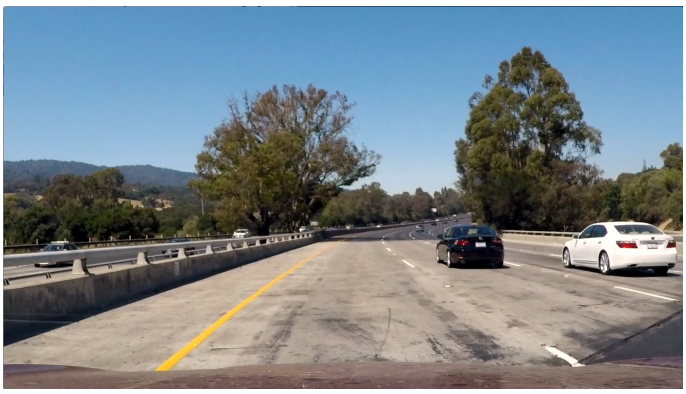
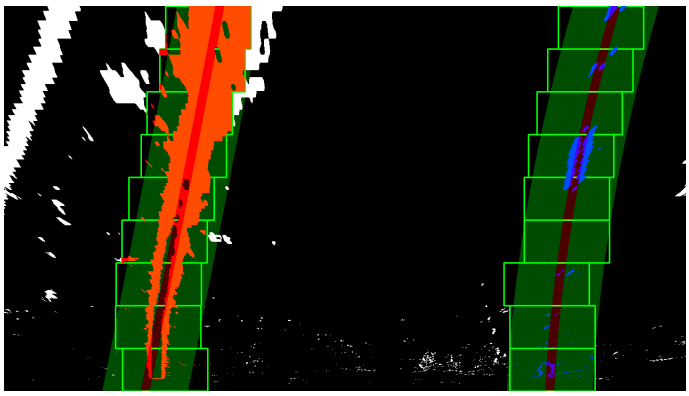
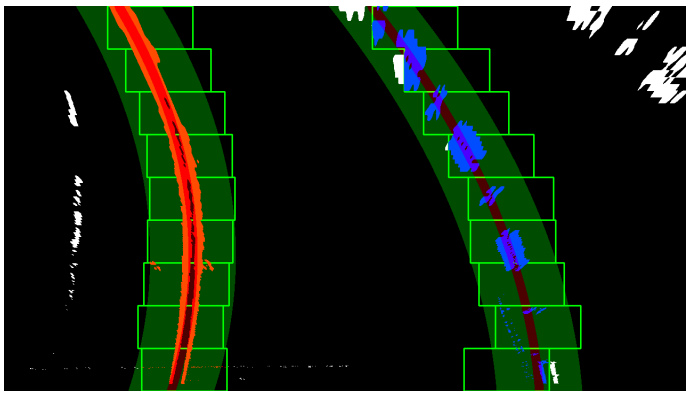
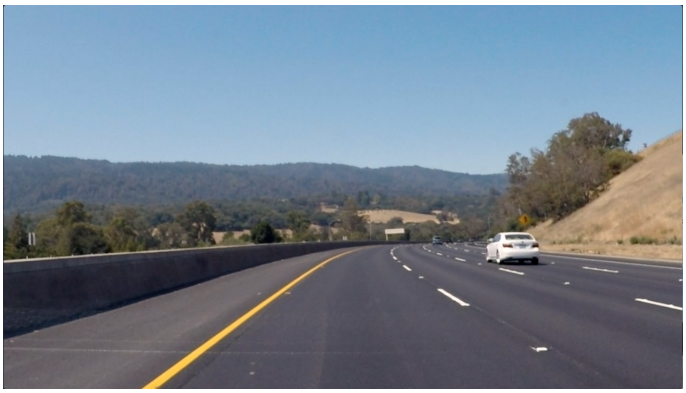
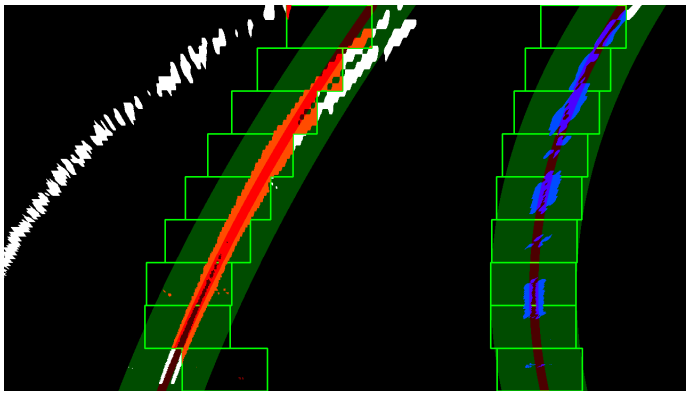
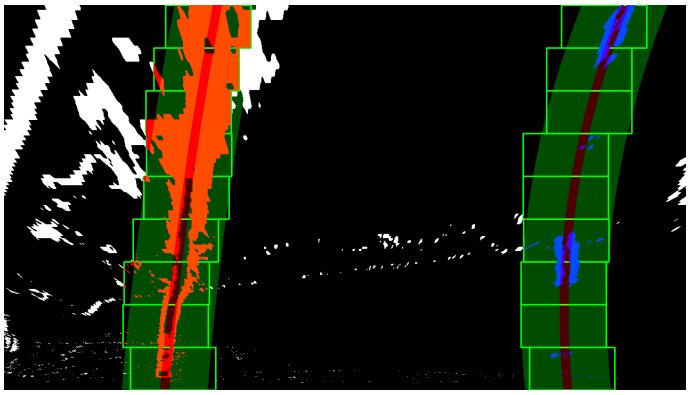
All of the points within the window for left lane are considered for fitting a polynomial of second degree for the left lane (same is done for right lane)

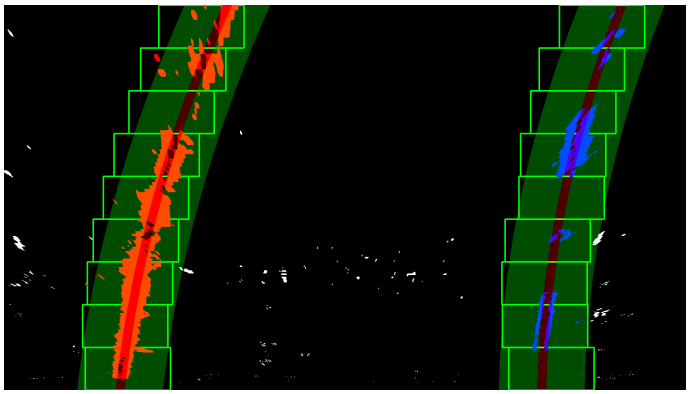
The code for creating perspective transform is contained in cell 18 of the IPython notebook located in "./Advanced Lane Lines-v02.ipynb"

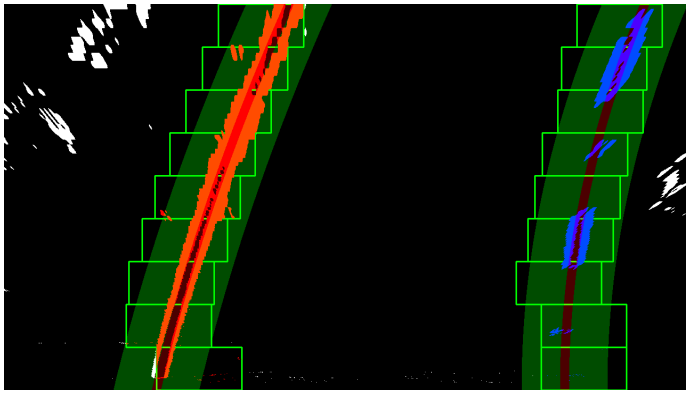
Function getLeftAndRightTracks contains this code.

Function fitPoly is used to fit the polynomial through these points

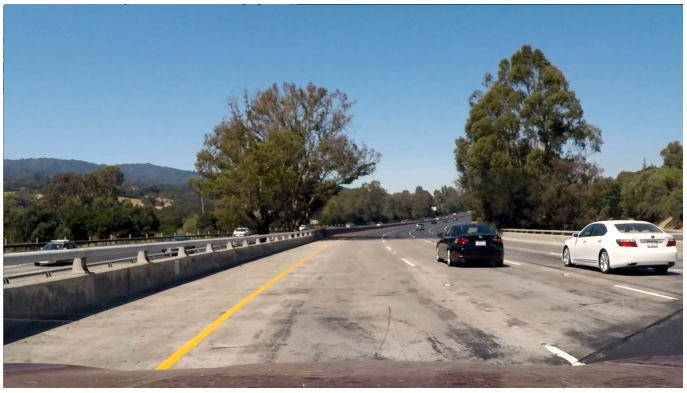
Following images show the polynomial fits

Original Image	Binary Warped and Polynomial Fit
	
	
	
	



Following images show the drawn Lanes

Original Image	Lanes drawn
	
	

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

It is calculated in function findRadiusocCurvatureAndLocation in code cell 33 of the IPython notebook located in "./Advanced Lane Lines-v02.ipynb

Calculation of Radius of Curvature and Vehicle position:

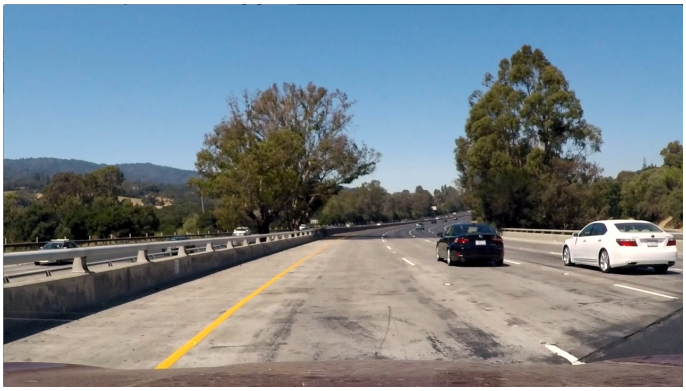
- First values of y axis(yvals) are taken in range from 0 to 720(image height)
- Assumed length in meters per pixel
    - y length in meters per pixel = 30/720
    - x length in meters per pixel = 3.7/700
- leftx and lefty points(on left lane) are transformed in meters (same is done for rightx
  and righty points on right lane)
- A line fit is found throught these transformed points
- Corresponding to the yvals in step 1, leftfit x points are identified (same for right lane)
- Average of the left and right x points is calculated to find average value for the curve
- A polyfit is done through these x points and corresponding yvals
- Radius of curvature is calculated using the formula
    f(y)=Ay^2+By+C
    Rad = (1 + (dx/dy)^2)^1.5)/(abs(second derivative of x wrt y)
- For camera position - bottom x value of left lane and right lane is found (calculation is in meters)
- average of this x value is calculated
- half of the image width is subtracted from this average value to find the center offset

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

I implemented this step in code cells 33,34 and 43 of the IPython notebook located in "./Advanced Lane Lines-v02.ipynb" .

Here are examples of my result on a test image:

Following images show the drawn area between the identified lanes

Original Image	Lanes drawn
	


Radius of Curvature = 289.946(m)
Vehicle offset : 0.103m left of center


Radius of Curvature = 402.929(m)
Vehicle offset : 0.079m left of center


Radius of Curvature = 784.717(m)
Vehicle offset : 0.211m left of center


Radius of Curvature = 652.565(m)
Vehicle offset : 0.139m right of center

Pipeline (video)

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!).

Here's a link to my video result (./movie.mp4) Video is added in the zip file also.

Discussion

1. 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?

I started with Camera Calibration and removing distortion from the test images. I then started work on perspective transform and tried various options.

I found out that selection of src and dest points is one of the crucial for the successful run of this program.

Then I tried to create binary image - I tried x gradient, y gradient alone, then along their magnitude gradient and then their direction gradient. It was only partially working. I tried various values for the thresholds. Then I tried color gradient also and chose S,L and V channels. I tried various combinations of these channels with value gradients. This has also been very crucial for the identification of lanes.

I noted that proper combination of perspective transform inputs and value and color gradients are the key to this problem. But this is still a basic problem and its solution. My solution does work partially for challenge video but it needs a lot of work to be done for harder challenge video. More work is required to be done for the fast changing curvature of the lanes. Better solution is needed for the curve fittings.

Current solution will likely fail in case of sharp curves and fast changing lane curvatures, various shadows and road distortions(damages , road repairs with various color gradients in repairs). More robust line fittings and more complex gradient function is needed to handle multiple situations.

Hard coding relating to src and dest points for perspective transforms may need to be changed.