SDCND Advanced Lane Finding

Matthias Schinacher matthias.schinacher@googlemail.com

2019-03-11

Contents

1 Intro

The project is a homework assignment for Udacity's **Self Driving Car Nano Degree**. The project is the second one called *Advanced Lane Finding*.

1.1 Implementation

I choose to implement the necessary code as a series of python scripts invoked from the command line, rather than implementing a notebook. I did use however the example `example.ipynb` as a starting point for the calibration.
I also used my versions of the solutions to the various quizzes in the course as a code base for my scripts.
The scripts implement each a specific part of the required task for the project. Those scripts that produce results used in later stages, use the standard python *pickle* module to save data as python objects.
The scripts have positional command-line parameters.

2 Calibration

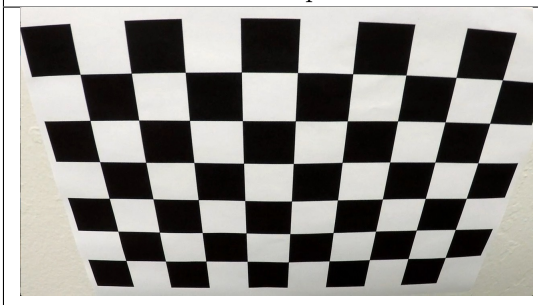
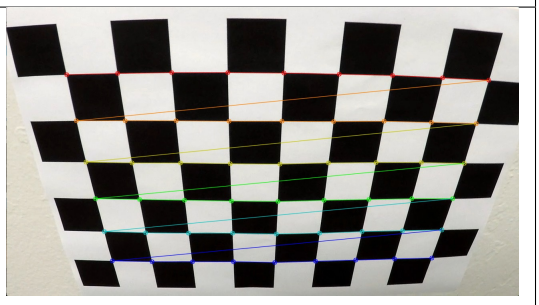
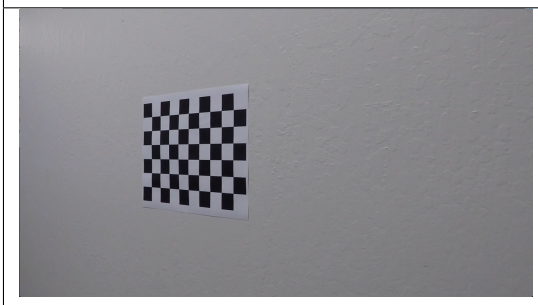
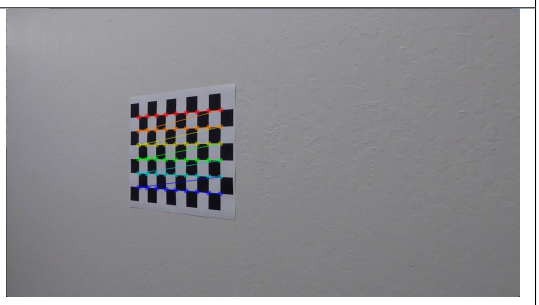
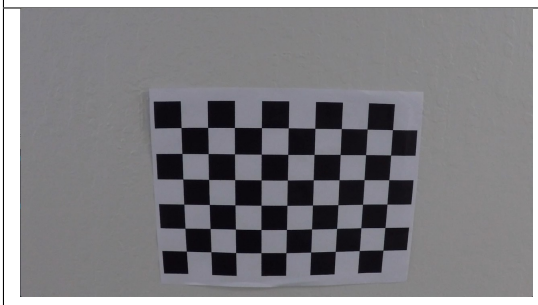
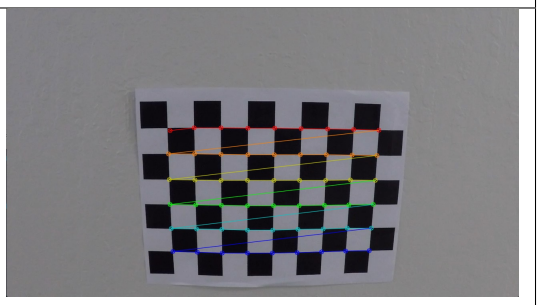
I calibrated the "camera" with the script textttcalibrate.py. It uses the `cv2` method `calibrateCamera(...)` to do the actual calibration and `findChessboardCorners(...)` to find the image points required within the given chessboard calibration pictures.
The calibration matrix etc. computed is written to a pickle file and the scripts allows to save and/or show modified calibration images, onto which the actual chess board corners (`findChessboardCorners(...)`).

Script parameters :

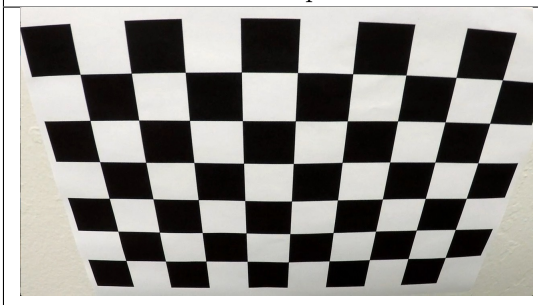
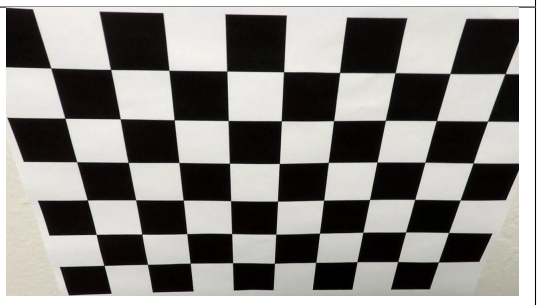
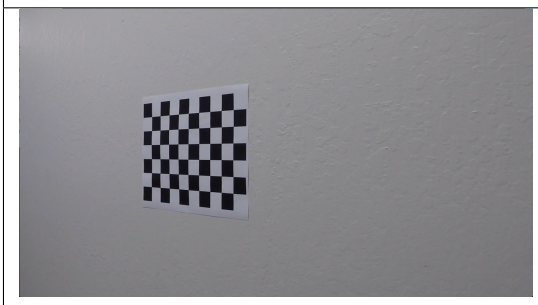
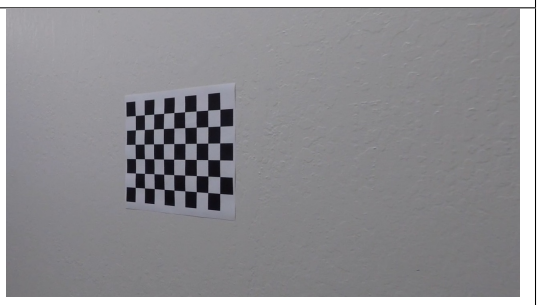
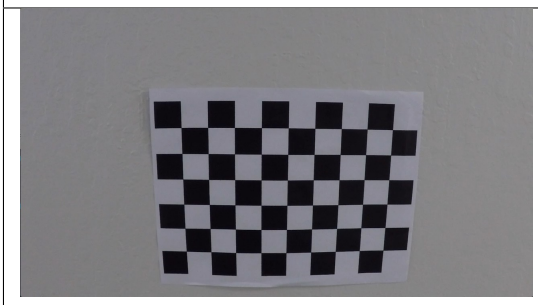
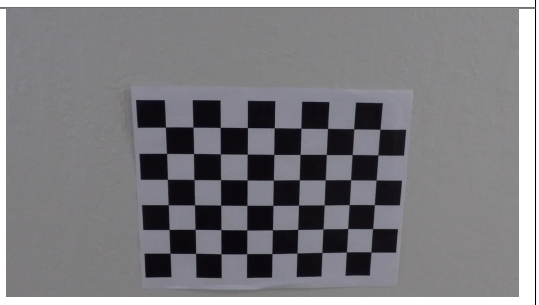
Position	Name	Description	Default
1	`show-flag`	show the modified calibration images?	`False`
2	`save-flag`	save the modified calibration images?	`True`
3	`pickle-file-name`	picke file name	`calibration.pickle`

(the "save-flag" is only for the pictures, the pickle file will always be written)

Graphs *(some examples of the chessboard corners found in the calibration images)* :

Calibration picture	... with corners identified
	
	
	

(same examples with undistorted images) :

Calibration picture	... with corners identified
	
	
	

3 Perspective transform

To find a good perspective transform I implemented the script `transform.py`; It takes as positional input parameters the x and y coordinates for a source quadrilateral and a destination rectangle defining the desired transform.
Additional parameters control whether to show the test images and how they transform on the screen and potentially save transformed images and the transformation parameters (in a pickle file).
This allowed me to experiment with a number of different transformations quite rapidly. The "best" transform I came up with *(dubbed "MA")*, was derived from the first picture *with straight lines*.
To get the coordinates for the source quadrilateral I would usually open one of the *straight lines*- pictures with gimp to identify 4 corners that were not a rectangle but would obviously be close to one in the real. I would then experiment with a couple of target rectangle coordinates.

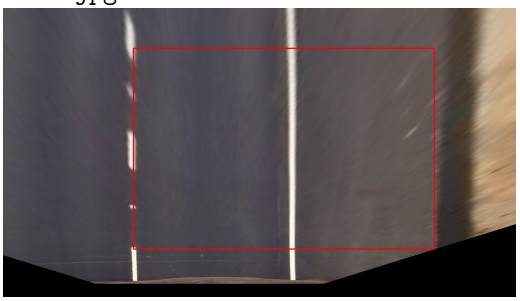
Script parameters :

Position	Name	Description	Default
1	`transformation-name`		(*mandatory*)
2	`cooordinate-string`		(*mandatory*)
3	`show-flag`	show images?	`False`
4	`save-flag`	save images?	`False`

The coordinates must be given as 16 integers within one string; that requires the string to be in quotes on the command line, like:

```
python transform.py MA '425 570 1276 570 806 458 586 458 325 600 1076 600 1076 100 325 100' 1 1
```

Graphs *(some examples perspective transform; source to transformed picture with the areas from the transform definition marked.)* :

source	transformed
`straight_lines1.jpg`	
	
`straight_lines2.jpg`	
	
`test3.jpg`	
	

4 OTHER

TODO: Rest of the report