Team: Project 42

SW Design Specification

Program to understand the movement of car from a target video

Team:	U1610041	Azizbek Kobilov
	U1610016	Akmal Karimov
	U1610047	Bekhzodkhon Makhkamov
	U1610052	Boburjon Bahodirov

Contents

1. Introduction

This section gives a scope description and overview of everything included in this document. Also, the purpose of this document is described, and a list of abbreviations and definitions is provided.

1.1 Purpose

The purpose of this document is to give a detailed description of the requirements for the project. It illustrates the purpose and complete declaration for the development of the system. It also explains system constraints, interface, and interactions with other external applications. This document is primarily intended to be proposed to a Professor for its approval and a reference for developing the first version of the system for the development team.

1.2 Scope

Program to understand car movement can be used in various big projects, e.g., in autonomous car project as a module to help the main algorithm of an autonomous car. It uses streaming video from a camera placed on the front side of the car as an input and outputs processed video with detected lane lines.

Moreover, the program can be enhanced and used out of autonomous car projects. Let's talk about one of its applications. If we place a camera and install the program to any car, it can stop the car or alert driver if he will fall asleep. In this case, if the driver falls asleep, his car will leave the road. It can be detected by analyzing road lane lines.

1.3 References

https://www.youtube.com/watch?v=gWK9x5Xs_TI

https://towardsdatascience.com/carnd-project-1-lane-lines-detection-a-complete-pipeline-6b815037d02c#.hcsizymg8

https://www.youtube.com/watch?v=eLTLtUVuuy4

https://github.com/topics/lane-lines-detection

Main source:

https://www.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

and repositories of this nanodegree program course.

Also, slides and online video materials provided in lectures.

1.4 Overview

The remainder of this document includes four chapters.

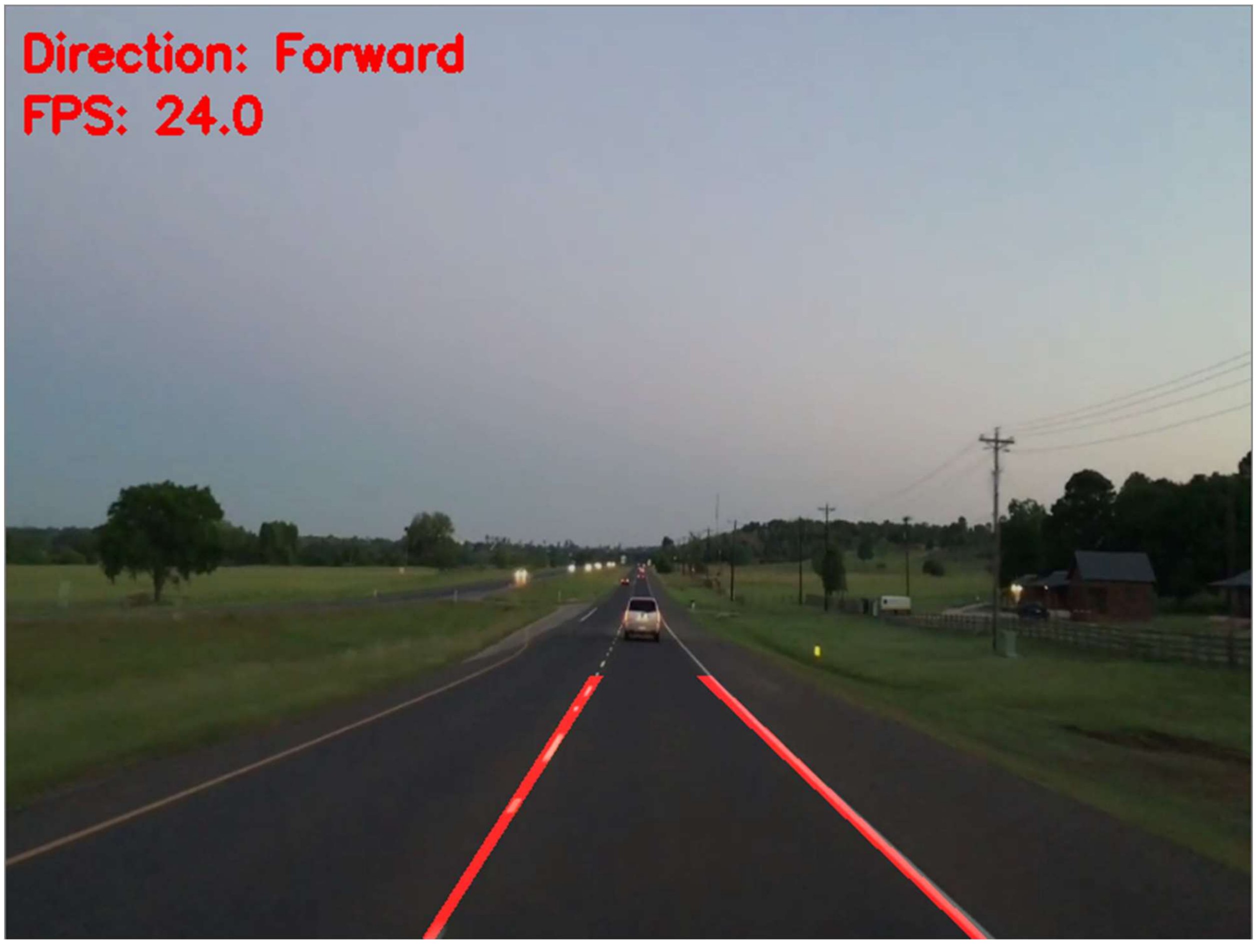
The second one provides an overview of the system functionality, the system constraints, and assumptions about the product.

The third chapter provides the requirements specification in precise terms and a description of the different system interfaces. Different specification techniques are used to specify the requirements more precisely for different audiences.

The fourth chapter deals with future plans and suggestions for development.

2. General Description

This section gives an overview of the entire system. The system is explained in the context of how the system interacts with the real world, and present the basic functionality. Finally, system limitations are presented.



Video frame (1024x768)

2.1 Product Functions

The program analyses the streaming video from the camera placed on the front of the car. It uses various image processing techniques to extract useful data. In the end, a user gets a result video with detected road lane lines.

2.2 Operating Principle

As was said in the previous sections, the program gets a video as an input. Full workflow:

- Video divided into frames, resized and each frame passed to processing.
- Frame is cut in half horizontally and bottom part passed further.
- Frame is a 3-channel image, so we convert it to one channel image by taking its grayscale.
- To remove noise, we use a Gaussian blur technique to blur the image using a 3x3 kernel.
- On a blurred image, using the Canny edge detection method, we get edges with low threshold 50 and high threshold 150 values.
- Once edges detected, we apply a mask with ROI (region of interest). ROI is static, hardcoded for each 1024x768, 800x600, 640x480, 400x300 dimensions. The reason for applying the mask is to remove all redundant lines from the image and leave only lane lines.
- Frame passed to HoughLinesP method from opencv-python module; result is all hough lines on the image
- If no lines detected, we assign lane lines from the previous frame. This fixes the case when there is no lane line in the video.
- Lines are separated into left and right, depending on their slope. Lines with positive slope, right lines, lines with a negative slope, left.
- Finding a median of slopes and medians of intercepts (left and right separately) gives us two solid lines.
- Then, we try to make lines smoother by taking a median of lines from the current frame and previous. If there is no line in the current frame, this function shall fix it by replacing the absent line, with a line from the previous frame.
- Lines are drawn on an empty image with the same shape as the original frame.
- Then, the image with lines, is masked in the same way as before. ROI is again, static.
- We extend dimensions of the image with lines to 3 channels, while it was only one channel in the beginning.
- Finally, the image with lines and original frame (before cutting) is blended using addWeighted function from opencv-python module.
- Using the slope of left and right lines, it is predicted, which side car is turning.
- Processed frame is streamed to the window or saved to the video file.

2.3 Constraints

As the project tells itself, the primary constraint is lane lines. Mountain roads or other types of roads without lane lines are out of the scope of this project.

Parts of the road are without lane lines shall cause troubles. A pasting line from the previous frame partially solves the problem. So, in our case, program copies lines from previous frame, if it can’t find lines in current frame. **But lines will exist even on the road without lane lines.**

Another problem is fixed ROI. We hardcoded ROI coordinates for all of the required dimensions. It would be better if lines are detected and masked without static ROI.

3. Specific Requirements

This section contains all of the functional and quality requirements of the system. It gives a detailed description of the system and all its features.

3.1 Hardware Requirements

Any type of microcomputer which can run python is enough. The more processor is powerful, the faster processing shall be.

If the program is used for a real-life situation, without offline video, a camera on the front of the car is required.

3.2 Program Components:

Program is divided into 3 modules:

- **main.py** – extracts the frames from the video and passes to the processing module. Also, it can save collect processed frames into one video and save it. Also, FPS calculated and written of processed frames.
- **classes.py** – keeps Video and Line classes that are required for making it easier to work with lines (calculating slopes and biases) and video files.
- **utils.py** – a place where all processes run. It keeps all functions and variables used for image processing.

3.3 Software Requirements

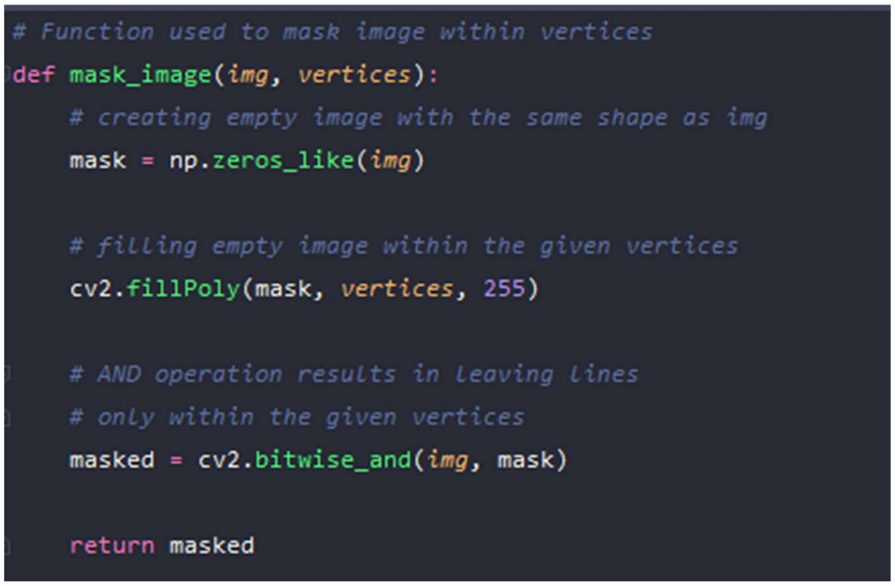
- Python 3.7.*
- python-opencv 4.2.*
- numpy 1.18.*
- time library
- ffmpeg (optional, for breaking videos into chunks)

Other dependencies (can be found in requirements.txt in the project folder):

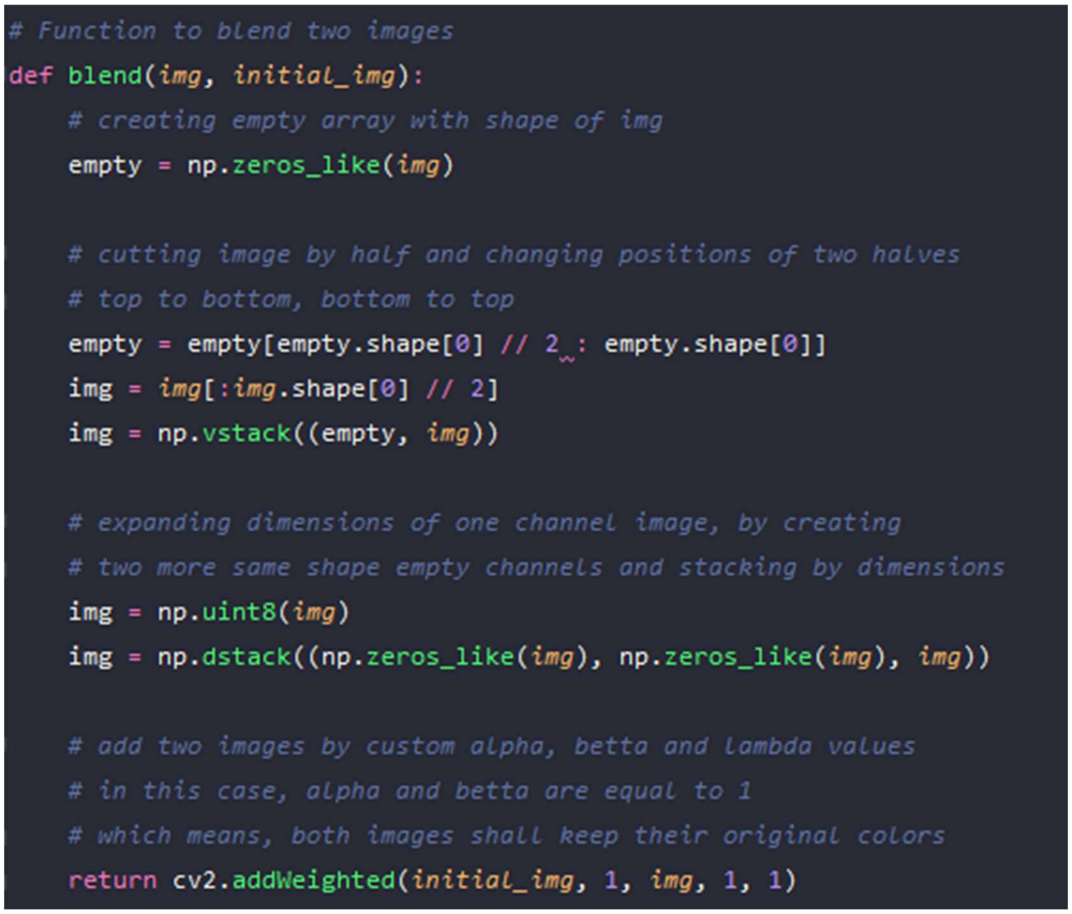
- cycler==0.10.0
- entrypoints==0.3
- importlib-metadata==1.5.0
- ipykernel==5.1.4
- ipython==7.13.0
- ipython-genutils==0.2.0
- jedi==0.16.0
- json5==0.9.4
- jsonschema==3.2.0
- jupyter-client==6.1.3
- traitlets==4.3.3
- jupyter-core==4.6.3
- kiwisolver==1.2.0
- pywin32==227
- pyzmq==18.1.1
- wcwidth==0.1.9
- matplotlib==3.2.1
- parso==0.7.0
- pickleshare==0.7.5
- prompt-toolkit==3.0.4
- Pygments==2.6.1
- pyparsing==2.4.7
- pyrsistent==0.16.0
- python-dateutil==2.8.1
- six==1.14.0
- tornado==6.0.4
- webencodings==0.5.1
- wincertstore==0.2
- zipp==3.1.0

3.3 Code Design (Example)

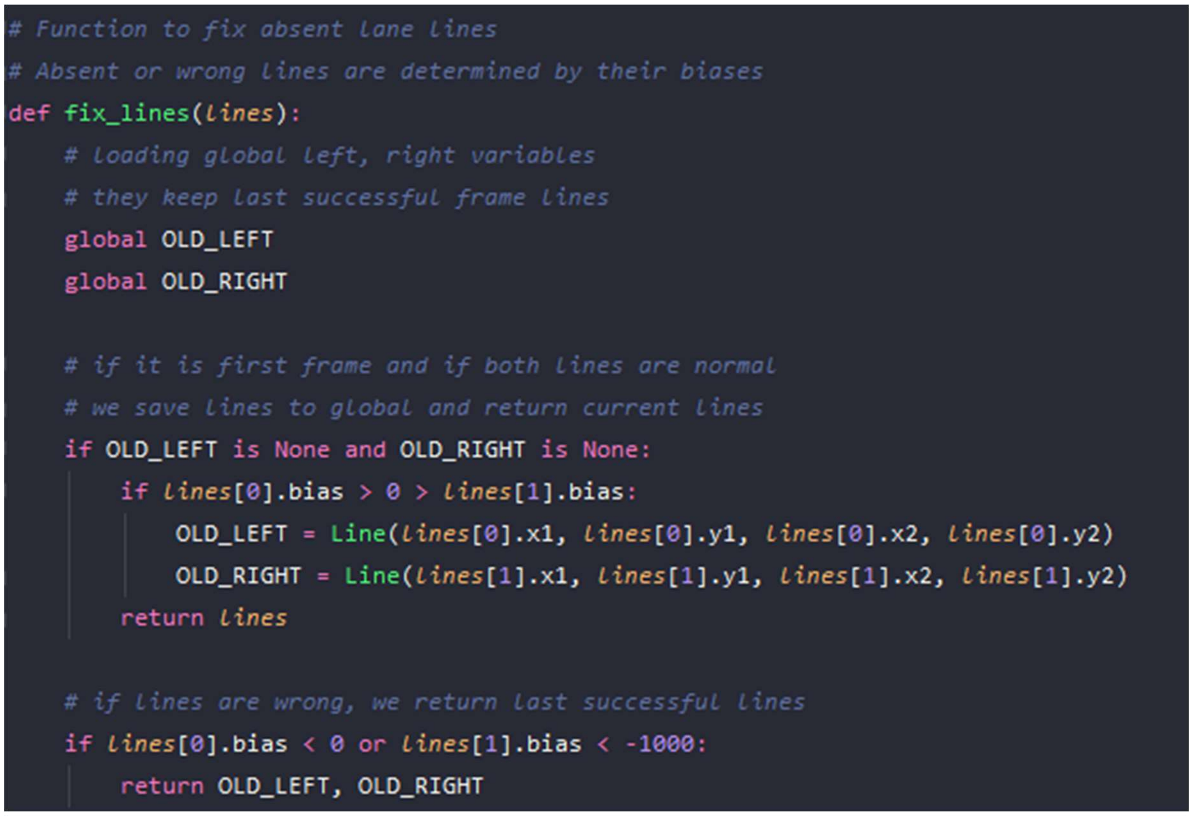
In this part, you can see code chunks with explanations.



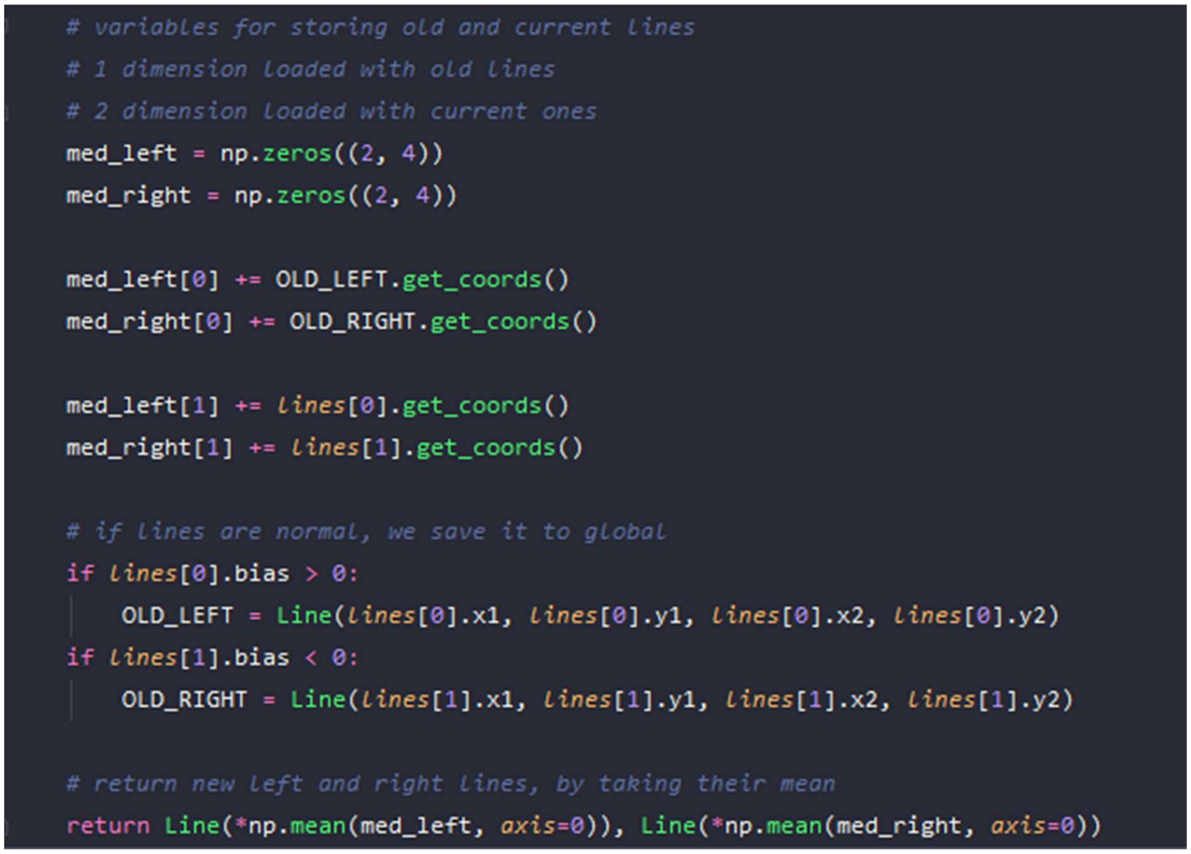
```python
# Function used to mask image within vertices
def mask_image(img, vertices):
    # creating empty image with the same shape as img
    mask = np.zeros_like(img)

    # filling empty image within the given vertices
    cv2.fillPoly(mask, vertices, 255)

    # AND operation results in leaving lines
    # only within the given vertices
    masked = cv2.bitwise_and(img, mask)

    return masked
```



```python
# Function to blend two images
def blend(img, initial_img):
    # creating empty array with shape of img
    empty = np.zeros_like(img)

    # cutting image by half and changing positions of two halves
    # top to bottom, bottom to top
    empty = empty[empty.shape[0] // 2 : empty.shape[0]]
    img = img[:img.shape[0] // 2]
    img = np.vstack((empty, img))

    # expanding dimensions of one channel image, by creating
    # two more same shape empty channels and stacking by dimensions
    img = np.uint8(img)
    img = np.dstack((np.zeros_like(img), np.zeros_like(img), img))

    # add two images by custom alpha, betta and lambda values
    # in this case, alpha and betta are equal to 1
    # which means, both images shall keep their original colors
    return cv2.addWeighted(initial_img, 1, img, 1, 1)
```



```python
# Function to fix absent lane lines
# Absent or wrong lines are determined by their biases
def fix_lines(lines):
    # loading global left, right variables
    # they keep last successful frame lines
    global OLD_LEFT
    global OLD_RIGHT

    # if it is first frame and if both lines are normal
    # we save lines to global and return current lines
    if OLD_LEFT is None and OLD_RIGHT is None:
        if lines[0].bias > 0 > lines[1].bias:
            OLD_LEFT = Line(lines[0].x1, lines[0].y1, lines[0].x2, lines[0].y2)
            OLD_RIGHT = Line(lines[1].x1, lines[1].y1, lines[1].x2, lines[1].y2)
        return lines

    # if lines are wrong, we return last successful lines
    if lines[0].bias < 0 or lines[1].bias < -1000:
        return OLD_LEFT, OLD_RIGHT
```



```python
    # variables for storing old and current lines
    # 1 dimension loaded with old lines
    # 2 dimension loaded with current ones
    med_left = np.zeros((2, 4))
    med_right = np.zeros((2, 4))

    med_left[0] += OLD_LEFT.get_coords()
    med_right[0] += OLD_RIGHT.get_coords()

    med_left[1] += lines[0].get_coords()
    med_right[1] += lines[1].get_coords()

    # if lines are normal, we save it to global
    if lines[0].bias > 0:
        OLD_LEFT = Line(lines[0].x1, lines[0].y1, lines[0].x2, lines[0].y2)
    if lines[1].bias < 0:
        OLD_RIGHT = Line(lines[1].x1, lines[1].y1, lines[1].x2, lines[1].y2)

    # return new left and right lines, by taking their mean
    return Line(*np.mean(med_left, axis=0)), Line(*np.mean(med_right, axis=0))
```

Discussions and Future Plans

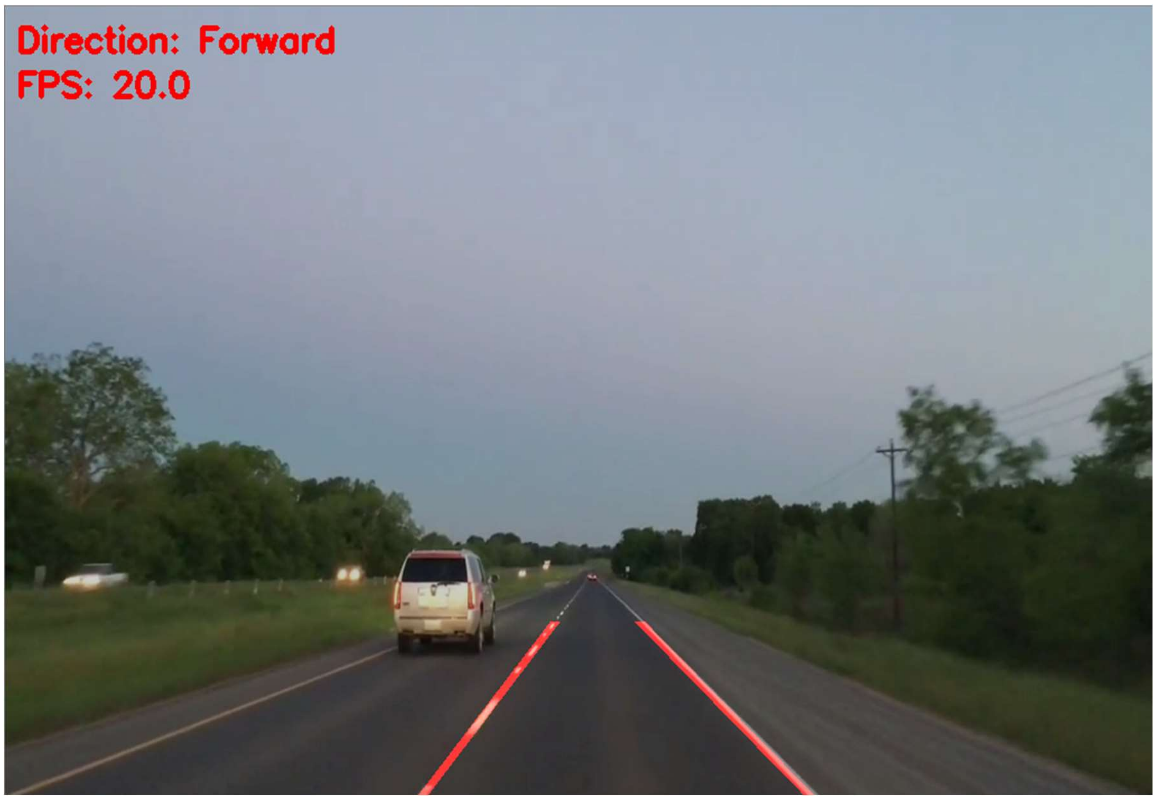
As it is said in the requirements:

You will have to predict the turn and calculate the frame per second (FPS) rate of your video. Video size should be chosen as: 1024x768, 800x600, 640x480, 400x300 and FPS should be compared.

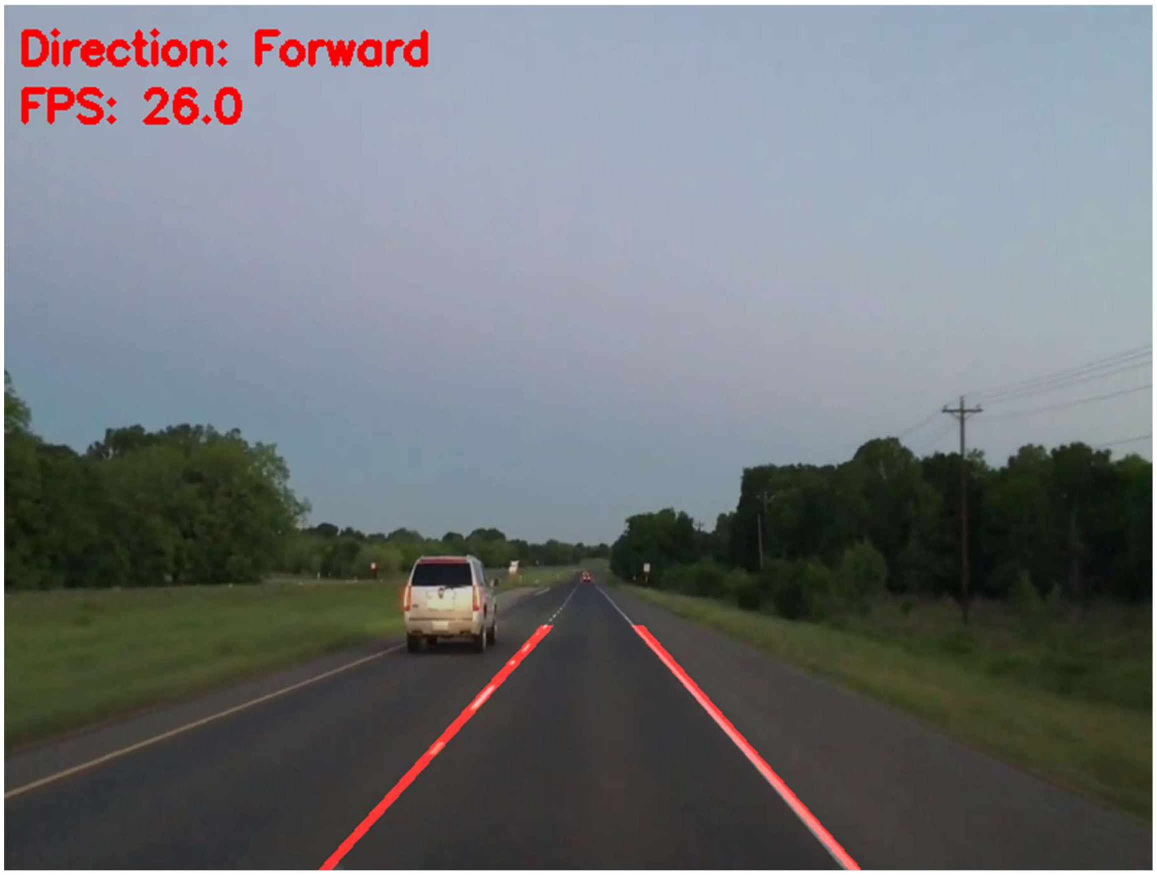
We have conducted tests on the laptop with 2.4GHz Intel Core i5 processor.

Results:

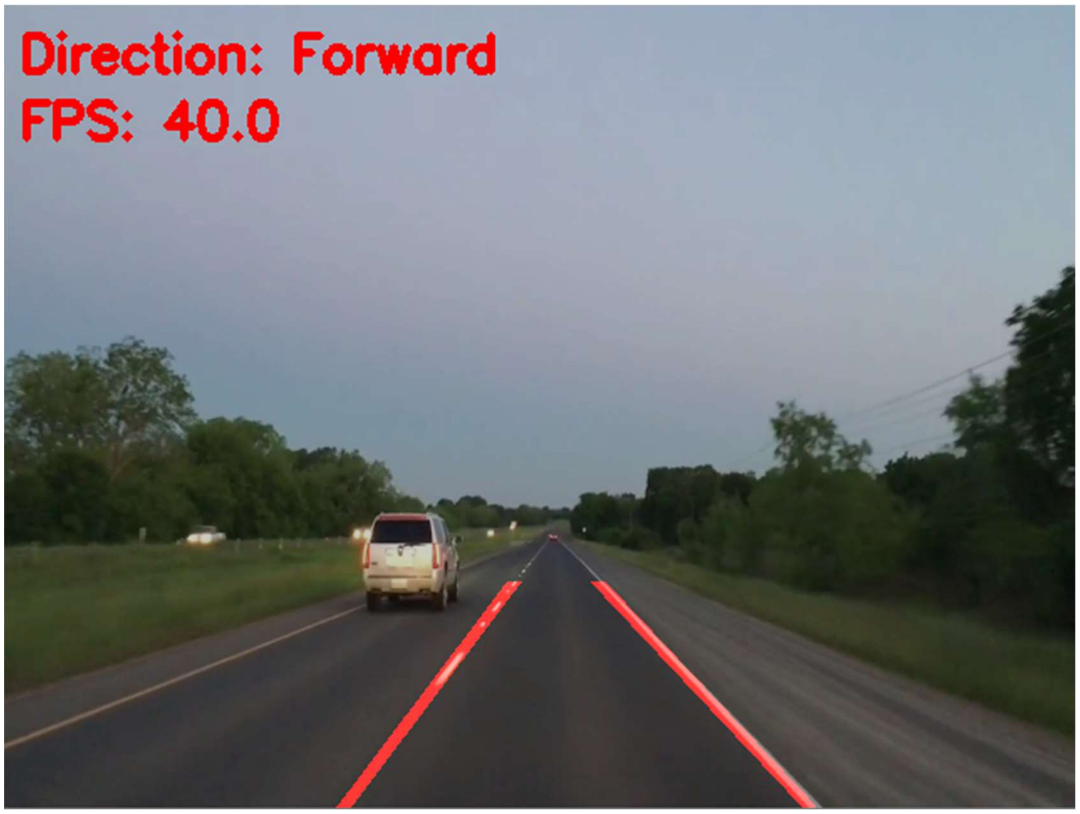
- 1024x768 -> 20~ FPS



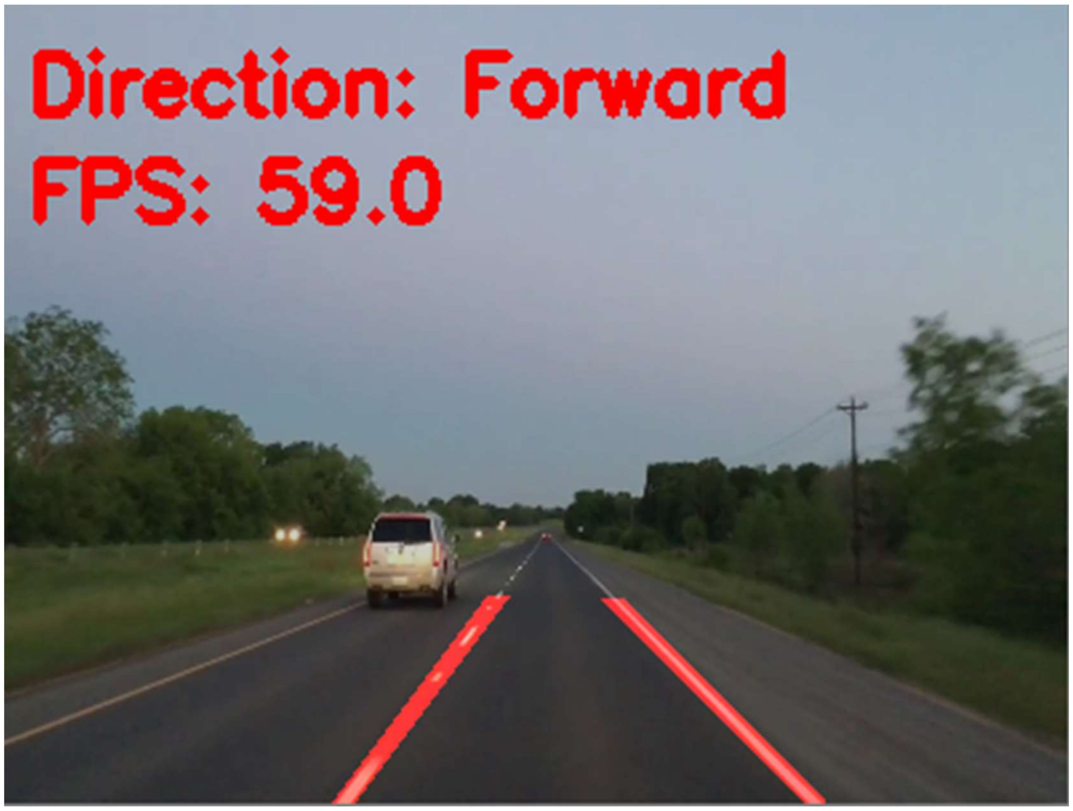
- 800x600 -> 26~ FPS



- 640x480 -> 40~ FPS



- 400x300 -> 59~ FPS



Turn prediction was accomplished by calculating slopes of lines and based on these values, program decides, where car is moving. Sometimes, it can show wrong side, e.g. when car is moving forward, but car is changing position inside its lane. In this case, camera perspective in relation to road lines, is changed a bit, so in the end, it changes turn prediction results.

All used sources can be found on the References page. No code was stolen. We have learned from other courses, as well as from open source repositories. The main source for learning was **Udacity Autonomous car Nanodegree program**, where the first module is about detecting lane lines.

If we develop the program further, we can add deep learning algorithms and models for lane line prediction. Also, this can be useful for detecting lane lines without masking all redundant lines from the image.

Project can be found at https://github.com/Tessium/road-lane-lines-detection.