A system for evaluation of human driving based on IoT and computer vision

1st Đorđe Janjić
Faculty of organizational sciences
University of Belgrade
Belgrade, Serbia
djordje.janjic@hotmail.com

***Abstract*—Object of research in this paper is human driving. The primary goal of this research is to simplify and improve driver education for newcomers. This is achieved through combination of technologies and libraries in Python. OpenCV is used for computer vision, which predicts and observes traffic participants via Raspberry Pi microcomputer and its camera. Haar Cascade is object detection algorithm that gives the ability to software to detect objects of interest for evaluating human compliance with traffic rules.**

Keywords—human driving, computer vision, artificial intelligence

I. INTRODUCTION

Safety of human driving, especially along side traffic participants in a busy traffic, depends on a large number of variables. Naturally, there will be some errors that can result in a tragic accident. Unfortunately, each year 1.35 million people are killed on roadways around the world, and crash injuries are among leading causes of death globally [1]. That's the main reason why companies like Tesla and comma.ai are working on automating vehicles, each on their own way.

Tesla is a leading company in this field, and they're making branded cars. Some of those cars have full autonomous option, which is known as autopilot. Their advanced sensor coverage with 12 ultrasonic sensors, 8 surround cameras and 360 degrees of visibility makes the car one of the safest choices regarding self-driving today [2]. In comparison, in Q1 2021, there was one accident for every ~ 6.74 million kilometers for drivers that had autopilot enabled, but NHTSA's data shows that only in US there is a vehicle crash every ~ 779.000 kilometers [2]. On the other side, there's also comma.ai, that tries to make everyday cars fully autonomous with their product, known as comma two. They are known for open source policy, so everyone can contribute to the openpilot software, and be a part of the community. This product can be really helpful during long trips, or simply as a way to rest the driver for some period, without worrying for his safety. Need for driving, while driver is exhausted, is another way to get into an accident. These are only some of the efforts to reduce traffic accidents, and fatal outcomes. However, not much is being done in educating newcomers, as it is as traditional learning experience, as it was in the last century.

This paper dives into the complexity of driving and human compliance with traffic rules. Main goal is to give drivers opportunity to visually see the results of their driving, which can be really helpful, especially for beginners. It evaluates compliance with traffic signs, road lines, and overall safe driving around other traffic participants, pedestrians, etc. After the driving session, drivers can evaluate their results, and on that basis to iteratively improve their driving.

II. METHODOLOGY

Using cameras and sensors with Raspberry Pi microcomputer [3] is a great way to improve the human experience and safety of an environment. Adding the logic to the software that uses the parameters from sensors and cameras, and ability to exchange data with other devices, makes the device in use a smart device. User is interacting with the device via user interface (UI), which is connected with the server side of the software that "communicates" with the data. On the other end, smart device with camera and sensors, communicates directly with the server side, so that parameters can be evaluated by the software and then be stored into the database.

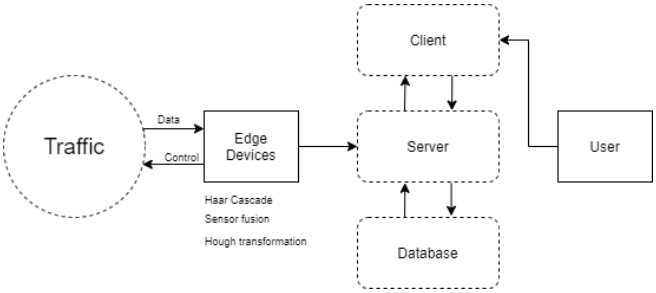

Figure 1 - System architecture

In this paper, three main components are identified as major points of interest:

- Management of distance from traffic participants
- Staying in lane
- Detecting traffic signs

A. Management of distance from traffic participants

One of the ways, how human driving can lead to an accident, is poor management of distance from other cars or ignoring pedestrians close to the pedestrian crossing. A way to fix that is detecting and tracking the cars and pedestrians while driving, which is possible with OpenCV library in

Python, and using camera for evaluating the distance and speed, which are the most important parameters for keeping the driver safe [4]. Common rule for safe distance is a "3-second rule", where driver can calculate the distance from another car using a fixed object that is passed by the car in front, and if he passes that object in less than 3 seconds, he is too close. That rule can be converted into distance from another car, but it varies based on the driving speed. If driver is driving at about 80 km/h, the safe distance would be at about 67 meters, and if the driving speed is at 50 km/h, then the safe distance is reduced to about 42 meters, etc. For purposes of this software, goal is to warn the driver if he is closer than 10 meters from the car in front of him. First, software detects cars in the frame via pre-trained Haar Cascade for car detection. Also, it needs a reference image of detected car for determining the focal length. Now that focal length has been found, we can calculate the estimated distance from the cars in front of the driver in all frames. If driver gets to close, screen flashes blue color and software warns the driver to be careful and to manage his distance in a text form in red color. For the entirety of the driving session, distance in centimeters will be shown in a text form in the top left corner of the frame.

Pedestrians are harder to detect and predict their behavior. Even in everyday human driving, it is sometimes hard to see some pedestrians in busy city traffic, and that can lead to serious and fatal outcomes. However, software can be trained to see things that human would normally miss. Goal is to detect and predict pedestrian behavior with pre-trained Haar Cascade for human detection, and based on that alert the driver if he's too close while not slowing down. The method for achieving this will be the same as in the car detection and distance estimation. Added feature is that if software detects zebra crossing and a pedestrian in its vicinity, it signals the buzzer to go off and alert the driver. Software detects zebra crossings via function used for line detection. That function is explained in section B.

B. *Staying in lane*

This tends to be difficult for newcomers to driving, but it can have some serious consequences. Sometimes, driver can easily wander off, and without knowing start to change lanes, which can be dangerous if there are already other cars. To mitigate that behavior, camera is tracking lines throughout the whole driving session, and if the driver treads the line, it will alert him with the built-in sensor, and it will be shown in the results afterwards. The lines are tracked via Hough transformation, which works in harmony with canny edge detection from OpenCV. The way this works is that frame is forwarded to the function for line detection, where the road is masked from the frame, as we want from software to only look at that part of the whole frame. Canny is good for extracting most of the lines from a frame, but Hough transformation can detect straight lines, which is useful for our purpose. Detected lines are colored in blue, but everything else is a black frame. That way, the software knows if in the masked area, there is a blue color, the driver is going into another lane, and it signals the buzzer to go off. If the frame is completely black, the software signals the buzzer to turn off. That way the driver can be alerted at all times.

Also, line detection works well for the zebra crossing detection, only now the buzzer will go off if the zebra crossing is detected with the pedestrian in its vicinity. Zebra crossing is detected if the frame is colored blue more than black. That way, the software knows that the driver is in front of the lines that spread throughout his lane.

C. *Detecting traffic signs*

Another important aspect of human driving and overall safety for all traffic participants, is human compliance with traffic rules that can change based on the location, weather conditions, and other factors. Using Haar Cascade, software can detect traffic signs, and then evaluate if driver follows the rules via logic implemented in Python programming language [5]. For example, if there's a stop sign, sensors should give the information to the software if driver stopped on the sign, as it's expected. If software detects that driver did not stop in front of the stop sign, points are deducted in the final score. After the driving session, the results are shown to the driver for his further evaluation.

III. SYSTEM FOR EVALUATION OF HUMAN DRIVING

This section describes the development of the software part and how are the software and hardware integrated.

For the software part, which is a huge part of the whole ecosystem, it has built-in logic for car, lane and traffic sign detection. It works via camera that is attached to the Raspberry Pi.

Technologies that are used for developing this system are:

- Flask
- MySQL
- OpenCV
- Python

Flask is a framework that is used to build the web application, where user can view and evaluate his results after driving sessions. User can navigate with html buttons on a page, and choose if he wants to start the driving session, or view the results (Figure 2). If user chooses to start the app, the camera will turn on, and the driving session will be started. When the driver ends the session, he can press the stop button and the results will be saved in the mysql database, which is implemented with pymysql library (Figure 3).

User can also view all of his results, by pressing the button "Preview results", which will redirect the user to the results page, and show him the table with the result and the date, when the result occurred (Figure 4).

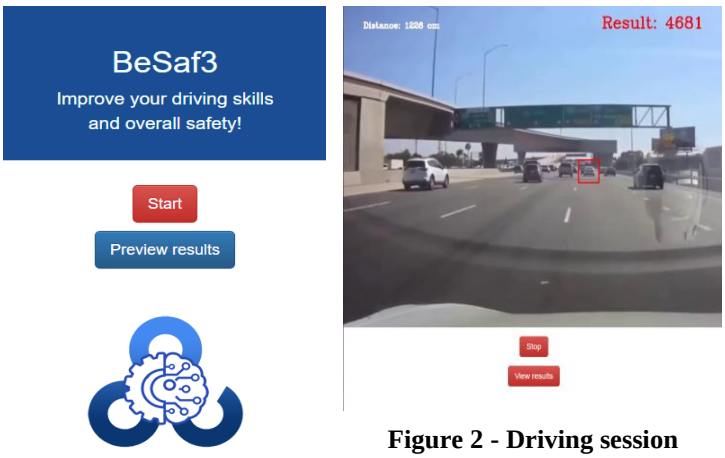

Figure 2 - Driving session

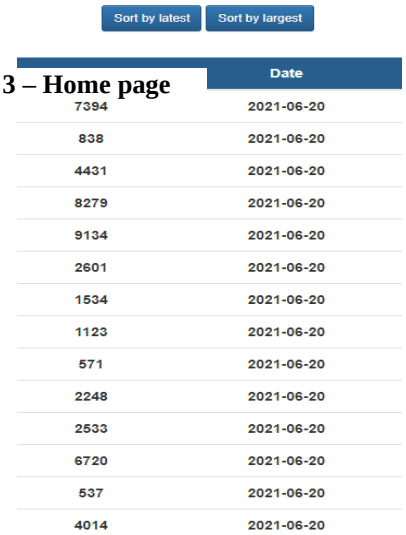

Figure 3 – Home page

Figure 4 - Results of driving sessions

For mobile application to store the results from driving sessions, some logic for stopping the camera must be implemented. The whole code for detection is stored in the main.py file, where the OpenCV library is used for detecting traffic participants (Figure 5).

The most important part of the software is car and line detection, with the distance management of the car in front of the driver. For distance management to work, we need to give the software a reference image to work with, and to find the focal length of the camera that is used for all the detection (Figure 6). When software detects the car, which is done through Haar Cascade Classification, it draws a red rectangle around it, and then it puts the distance in centimeters at the top left corner of a frame. That is being done with OpenCV putText function. If the driver is too close, in this case closer than 8 meters, the blue overlay is shown on the frame, and the warning text is displayed on the bottom part of that frame. Also the points of the result are being reduced by 15 (Figure 7).

Line detection also works within the OpenCV library, but with Hough Transformation function (Figure 8). First, mask is applied on each frame, where the region of interest is the road, as we need the software to look for the lines only on that part of the frame. The lines are colored in blue, and the rest of the frame is black. If the blue color shows up in the frame, that means that the driver is either too close, or hitting the line. If that's the case, then the software will signal the buzzer to go off, every time when the line is showing up in the frame, and the points are being reduces by 8.

One more functionality of the line detection, is the ability to detect zebra crossings. The logic is the same, just now the zebra detection depends on the blue color being the more dominant one in the frame. If that is the case, and the pedestrian is detected, buzzer goes off.

Pedestrian detection is the same as car detection, where Haar Cascade Classifier algorithm is used. Only now, the yellow rectangle is being drawn around pedestrians (Figure 9).

The third functionality of this software is sign detection, where only stop sign detection is implemented. It also uses Haar Cascade algorithm for detection. It draws purple rectangle around the sign, and the software is looking if the driver stopped while the sign is being detected (Figure 10). That is being done with comparing the previous part of the frame with the next ones. If that part of the frame is the same, that means that driver indeed stopped at the stop sign. If that is not the case, the points are being reduced.

```
    while(1):

        ret, frame = cap.read()


        if ret:

            global result
            result = result + 25

            print(result)

            #line_frame = lineDetector(frame)
            lineDetector(frame)
            carDetection = detectCars(frame)
            #cv2.imshow("CAR D", carDetection)
            carDetectionWidth = car_width_global

            #print("CAR D WIDTH",carDetectionWidth)
            #overlay = apply_color(carDetection, 0, 0, 0, 0)

            if carDetectionWidth != 0:

                Distance = distanceFinder(foc, know_width, carDetectionWidth)
                DistanceInCM = round(Distance)
                cv2.putText(carDetection, f"Distance: {DistanceInCM} cm", (50,50), fonts,0.6, (255
                cv2.putText(carDetection, f"Result: {result}", (650,50), fonts,1.2, (0,0,255),2)
                print("distance", DistanceInCM)
                if(Distance < safe_distance):
                    cv2.putText(carDetection, "WARNING! SLOW DOWN!", (400,650), fonts,1.2, (0,0,25
                    blue = 230
                    intensity = 0.3
                    result = result - 15
                    print("BLIZU KOLA, SMANJUJEMO REZ")
                    carDetection = apply_color(carDetection, intensity, blue, 0, 0)
            else:
                cv2.putText(carDetection, f"Not available", (50,50), fonts,0.6, (255,255,255),2)

            # formatiraj za prikaz u pretrazivacu
            imgencode = cv2.imencode('.jpg',carDetection)[1]
            strinData = imgencode.tostring()
            yield (b'--frame\r\n'b'Content-Type: text/plain\r\n\r\n'+strinData+b'\r\n')
```

Figure 5 - Looping through frames

```
safe_distance = 800
know_distance = 800 #centimetri
know_width = 150
fonts = cv2.FONT_HERSHEY_COMPLEX



def focalLength(measured_distance, real_width, width_in_rf_image):
    focal_length = (width_in_rf_image*measured_distance)/real_width
    return focal_length

def distanceFinder(focal_length, real_car_width, car_width_in_frame):
    distance = (real_car_width*focal_length)/car_width_in_frame
    return distance

def detectCars(frame):

    car_width = 0
    frame_g = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    #maskiranje
    vertices = np.array([[(550,600),(550, 300), (750, 300), (750,600)]], dtype=np.int32)
    mask = np.zeros_like(frame_g)
    cv2.fillPoly(mask, vertices, 255)
    masked_image = cv2.bitwise_and(frame_g, mask)

    cars = car_tracker.detectMultiScale(masked_image,1.3,5)


    global car_width_global
    for(x,y,w,h) in cars:
        cv2.rectangle(frame, (x,y), (x+w, y+h), (0, 0, 255), 2)
        #print("ww",w)

        car_width_global = w

    #print("cwg",car_width_global)

    return frame
```

Figure 6 - Car detection and focal length

Figure 7 - Warning text

```
def lineDetector(frame):



    frame_g = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    frame_blurred = cv2.GaussianBlur(frame_g, (7,7), 0)

    threshold_low = 10
    threshold_high = 100

    canny = cv2.Canny(frame_blurred, threshold_low, threshold_high)

    #vertices = np.array([[(10,400),(340, 250), (460, 250), (740,400)]], dtype=np.int32)
    vertices = np.array([[(430,510),(500, 450), (650, 450), (750,510)]], dtype=np.int32)
    mask = np.zeros_like(frame_g)
    cv2.fillPoly(mask, vertices, 255)
    masked_image = cv2.bitwise_and(canny, mask)

    rho = 2             # distanca u pikselima
    theta = np.pi/180  # ugao u radijanima
    threshold = 40     # min broj glasova
    min_line_len = 10  # min broj piksela za liniju
    max_line_gap = 80  # maksimalni razmak izmedju linija
    lines = cv2.HoughLinesP(masked_image, rho, theta, threshold, np.array([]), minLineLength=m

    # crna slika
    line_image = np.zeros((masked_image.shape[0], masked_image.shape[1], 3), dtype=np.uint8)

    if lines is not None:
        for line in lines:
            if line is not None:
                for x1,y1,x2,y2 in line:
                    cv2.line(line_image, (x1, y1), (x2, y2), [255, 0, 0], 20)

    mask = lineHit(line_image)

    suma = np.sum(mask)
    print(suma)
```

Figure 8 - Line Detection

Figure 9 - Pedestrian detection

Figure 10 - Traffic sign detection

For the hardware part, this system uses Raspberry Pi with the buzzer sensor for warning the driver if the line is hit, if he is too close to the car in front of him, or if the pedestrian is detected at the zebra crossing.

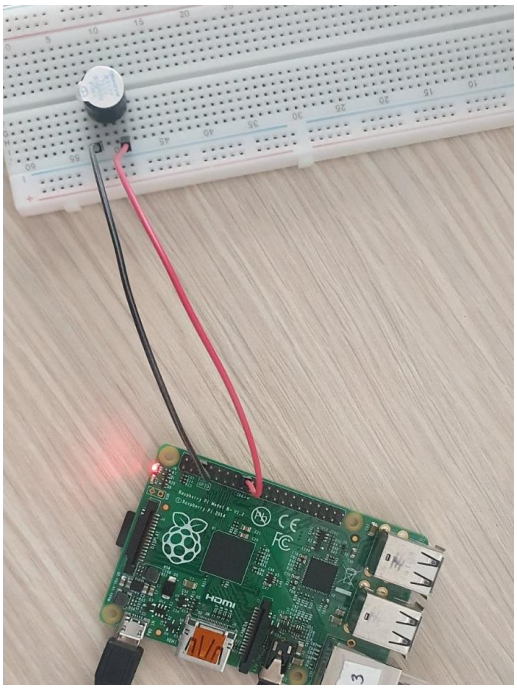

Figure 11 - Raspberry Pi with the buzzer

Fritzing scheme of how the buzzer is integrated with the Raspberry Pi:

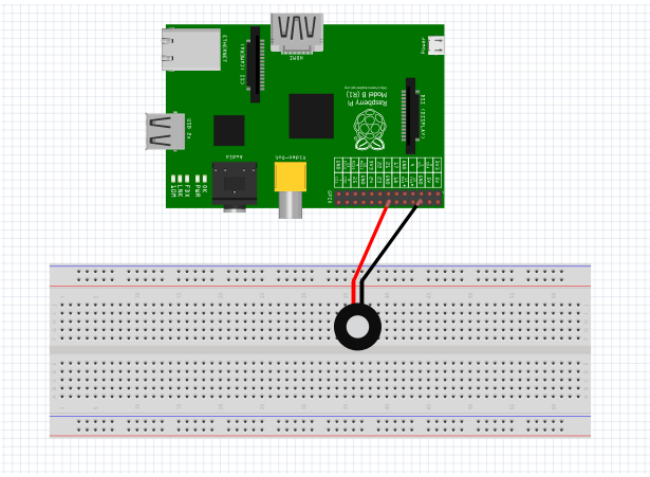

Figure 12 - Fritzing scheme

IV. Conclusion

This paper presents an application for improving the safety of a human driving experience. It shows the details of how some traffic objects are being detected, and then how the evaluation is being done based on the driving at that moment. The goal is to try and mitigate potential traffic accidents, but at the same time to make driving experience more enjoyable and fun, with the addition of the results of driving sessions.

The idea with the web application, is to make this system user friendly to the end users, and give them the ability to start the detection of the objects with just only one click, and to view the results of driving sessions in the same manner.

Future development of the software is going to include detection of more traffic signs, and refining the existing detection of traffic objects. Also, more sensors are going to be added to the end product, like LIDAR for detecting the quality of the road, for the ability to warn the driver if he needs to slow down. As for the web application, it's going to have the functionality of deleting the past results, and more detailed information of driving sessions, like where the driver made the error that resulted in reduced points. All of this will improve this system, and with that overall safety of a human driving.

References

[1] CDC, accessed 11 May 2021, < https://www.cdc.gov/injury/features/global-road-safety/index.html>

[2] Tesla, accessed 10 May 2021, < https://www.tesla.com/VehicleSafetyReport>

[3] B. Radenković, M. Zrakić-Despotović, Z. Bogdanović, D. Barać, A. Labus, and Ž. Bojović, Internet inteligentnih uredjaja. Fakultet organizacionih nauka, 2017.

[4] OpenCV, accessed 5 May 2021, <www.opencv.org>

[5] Ethan, ecd1012, Autonomous Driving Object Detection on the Raspberry Pi 4, accessed 9 May 2021, <https://github.com/ecd1012/rpi_road_object_detection>