WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: COMPUTER SCIENCE in English

BACHELOR THESIS

SUPERVISOR:
Conf. Dr. Cosmin Bonchis

INDUSTRY COORDONATOR:
Ing. Dr. Flavius Gligor

GRADUATE:
Paul Cvasa

TIMIŞOARA
2021

WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: COMPUTER SCIENCE in English

Traffic Participants Detection

SUPERVISOR:
Conf. Dr. Cosmin Bonchis

GRADUATE:
Paul Cvasa

INDUSTRY COORDONATOR:
Ing. Dr. Flavius Gligor

TIMIŞOARA
2021

Contents

Abstract

Nowadays we have a lot of traffic and in order to help reduce car accidents and the resulting human injuries, I have written a program that takes video feed from a camera mounted on a car and warns the driver of possible dangers that may occur based on the approximate distance from the traffic participants that exist in front of the ego vehicle. However I also have to account for its trajectory, as not every vehicle or pedestrian detected in the ego vehicle's field of view can be an immediate danger. It is basically an ADAS (Advanced Driver Assistance System) for older vehicles that don't have this function equipped from the factory. In order to achieve these goals I had to use Python 3.9 alongside the following libraries: Tensorflow, which is an open source platform for machine learning and has a Python API that I used to download and use a model for object detection; OpenCV, which I used to split my input videos into frames and analyze each frame to detect all traffic participants; NumPy, to expand each frame to match the model's expected shape and to also squeeze the arrays formed after the detection and PySimpleGUI, which is a Python library used for the graphical user interface.

Chapter 1

Introduction

The main purpose of this thesis problem consists of creating a smart enough system that provides multiple safety warnings and options so that we may live in a safer driving world in the future that we make for ourselves. The main planned feature is an efficient and reliable detection system that can track all of the traffic participants that are within your premises, such as cars, trucks, buses, motorcycles, bicycles and above all else, pedestrians.

My motivation to solve this problem was born when I saw that the majority of newer cars offer so many safety functions in terms of collision warnings, but a lot of older cars are currently still on the roads as not everyone has the money to buy a new one. So my idea was that everyone could have a spare old laptop and a webcam that can be used to run my application and provide a basic ADAS (Advanced Driver Assistance System) functionality for their older car.

The first related work that I found was UGV Driver Assistant[5], a dash camera app for Android/iOS, which also offers sign detection, road detection and distance control. The distance control doesn't offer a warning early enough to have a reaction, mainly because it uses the phone's hardware. It also drains the battery very quick, and the road detection isn't quite useful. The signs however work pretty good and a lot of Romanian traffic signs are still detected. The second app that I found is Drive Assist[7], available only for Android devices. However this is mainly in a 'proof of concept' state since 2016. It featured a traffic participant detection using only the OpenCV library. The website was taken down so I couldn't test its performance.

In this thesis I also presented the application's functionalities and capabilities that had to be met in order to be useful at least as a prototype for the future planned updates. This is one of the most important parts for the creation of the desired proof-of-concept for my program. Based on these functionalities I had to engineer the architecture of the application (including the use case, component and workflow diagrams) to create a compact but powerful software that is fast enough to withstand a busy day of traffic participants at the same time and also send the warnings in a relevant time frame for the driver to have a reaction and prevent the accident.

In the next chapter called "Related Work" I have presented the two above mentioned applications with implementation details and screenshots, along with the current development phase and compare them. In the "Problem Description" chapter I have written about the problem that this application will solve, and also about important information gathered from other related articles. Apart from the previous mentioned ones, I have added the program's functional and non-functional requirements. The "Author Contribution" chapter is composed of the implementation manual, which briefly presents how to system works, a use case and component diagram to understand the application architecture better. Another important information from this chapter is the user manual, presenting how to install and use the program. The chapter is ending with the system and software requirements to run the application. The last chapter called "Conclusions and Future Work" is mainly stating the benefits of using the program and the current planned updates and new features for it.

Chapter 2

Related Work

2.1 UGV Driver Assistant

2.1.1 Description

UGV Driver Assistant[5] is a dash cam app which has some advanced driver assistant systems (ADAS), such as: vehicle and pedestrian detection, road edge detection (example shown in Figure 2.1), road sign notifications (only for speed limits, stop signs, crosswalks and yield sign) and distance control warning. It also saves recordings when prompted by the user and the recordings are shown in a list that can be seen in Figure 2.2. Last update/info on Google Play: June 18, 2020. Last update/info on App Store: September 11, 2018.

2.1.2 Technical Details

The version used on the Google Play store was made in Android Studio using Java and Kotlin alongside the OpenCV library used for image detection and video processing. It also uses Tensorflow Lite library which has a set of tools that enables machine learning by running trained models on mobile devices, which supports the Android platform.

2.1.3 Screenshots

Figure 2.1: Example of Road Detection in the UGV app

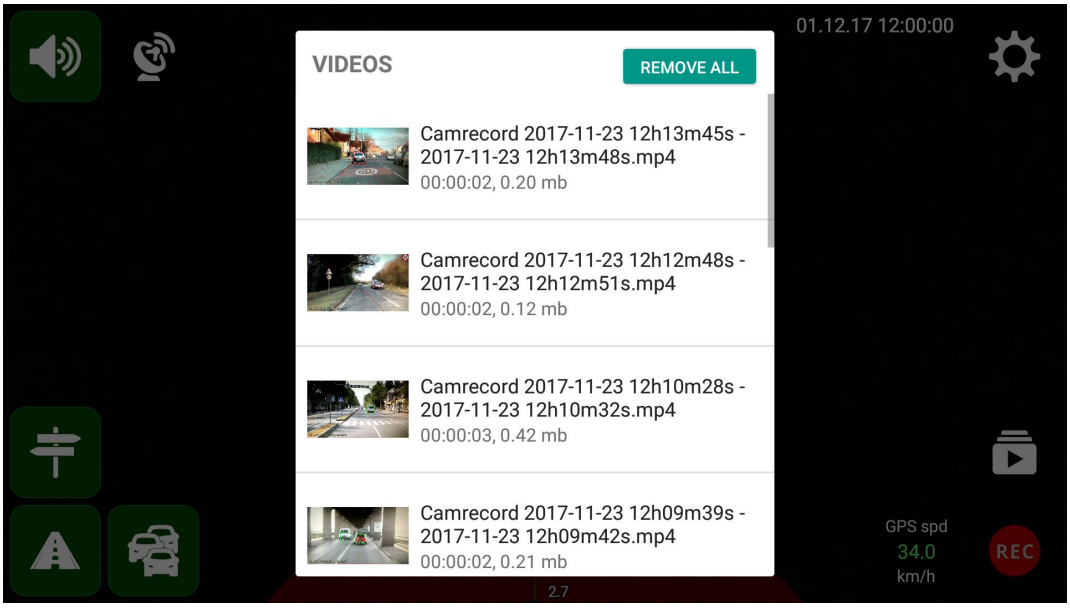

Figure 2.2: Recordings screen in UGV app

2.2 Drive Assist

2.2.1 Description

The Driver Assist[7] app was created in 2014 but it was abandoned afterwards (no more info or updates from the creators since 2016). It uses your smartphone's camera mounted on your vehicle windscreen in order to detect other cars, bicycles or pedestrians and alert imminent crashes. As for the accuracy of detection, it's still in the beta version and the website was deleted so it can't be tested. In Figure 2.3 we can see the app reacting to a pedestrian crossing the road. Figure 2.4 shows the computed distance to an oncoming car.

2.2.2 Technical Details

The algorithms are written in C++, but it uses JNI to connect with the Java language used by the Android device. This method improves the performance of the app because the code is written in a native language. As for the image detection part, it uses OpenCV library, which also supports Android NDK. It also computes the approximate distance towards the detected objects but it's not very accurate on finding all the traffic participants.

2.2.3 Screenshots

Figure 2.3: Example of a pedestrian crossing in the DriveAssist app

Figure 2.4: Example of a computed distance to oncoming vehicle in Drive Assist app

Chapter 3

Problem Description

The system's main purpose is to track every traffic participant that is detected in front of the camera and predict the ego vehicle's path, therefore providing an overview of possible dangers that can happen. With such a program we can avoid life-threatening injuries and accidents. That include those that can happen to pedestrians who might be either unaware of their surroundings or what might happen if they cross the street on a red light.

The system has to do the following tasks:

- Detect every traffic participant in front of the ego vehicle;
- Track every traffic participant in front of the ego vehicle;
- Know the ego vehicle path (to reduce false-positives warnings);
- Compute the approximate distance to the traffic participant in the path of the ego vehicle
- Send a warning if the distance gets too small

Another innovative feature is that of alerting the driver of a possible crash if the ego vehicle is too close to another traffic participant, which of course can be done with the help of the detection algorithms that are constantly running in my system. The feature is highly accurate and dependable because of its very complex trained model to detect objects, seeing similar patterns and alerting the user of everything that can happen before it's too late. Among other things, I also implemented a simple feature or mechanic such as being able to track the ego vehicle's current path; that way we can avoid all kinds of uneventful situations that can happen either on the opposite side of the road or on the right side of the road.

The system also comes with multiple quality of life features that can be adjusted to everyone's taste and needs, suited to match a specific person's criteria. These features range from having the option to specify the type of road in which the application is used, in order to have a precise warning sent. For example, a warning should be sent earlier distance-wise on a highway than in a city. My goal was to make this system available for a wide range of vehicles of multiple categories and sizes, securing a better future for everyone.

Besides all the functionalities that the system is able to do, mentioned above, I also tried to reach a high performance milestone in order to detect in time all the traffic participants.

The program is capable of excellent performance so that it can be useful and operable during high traffic situations and rush hours. Small errors might appear during the system's run time, for example a false-positive where a person from a billboard advert might be detected as a pedestrian, but nothing serious that could lead to the program crashing, so the software works and stays up-to-date with the reality as long as it's open, without interruptions. Limit wise, the system is able to handle all the traffic participants that are present in the system's camera field of view, with no exceptions.

The most important ADAS function is the FCW (forward collision warning) because most of the accidents in traffic happen when a driver fails to maintain a safe distance from the vehicle in front of him and that vehicle brakes unexpectedly to a full stop very fast. This situation leaves the driver unprepared because if the distance between the vehicles is too small, and the driver is either distracted by secondary tasks or simply it happens too fast for a braking maneuver to be possible, it will end with an unfortunate crash. However, there had been numerous rumours that if a car is equipped with such system, the driver will engage more in secondary tasks and won't pay enough attention to the surroundings, because of feeling safer that the system will save any occurring situations.

However, I found this study [10] where 30 drivers are tested whether a forward collision warning along with an automated braking system will encourage the drivers in secondary tasks because of feeling safer with the systems active. The article also states that a lot of other studies have shown how safety improves by using a front collision warning system, and the number of rear-end crashes decreases. It also states that the system can have an acoustic-visual warning but also only acoustic because it guides the attention of the ego vehicle driver to the dangerous and possible crash. Resulting in improving the reaction time of the driver that used the mentioned system. The FCW (Front Collision Warning) systems can aid the vigilant driver almost as well as the distracted one because the studies specified in the article stated that they had an improved reaction time when the system was used. Even though most of them showed an improvement and also a decrease in the severity of the accident, in some scenarios, the system wasn't able to predict early enough to mitigate the crash.

In order to understand the problem, I needed to know more about the driver's "gaze". This gaze behavior is produced by some missing expectations which happen on the road, for example drivers do not focus at all or not as long as it should when they do not expect that something might be happening there, so the detection period of time will increase in such unexpected situations. In order to achieve a safe driving behavior, the first step for the person in control of the vehicle is to perceive the scenario, next would be to understand it and the third and most important step is to foresee what could go wrong. When the scenario feels odd and the third step isn't done properly, the driver will begin to "gaze" away from

the vehicle in front, thus resulting in a lack of response time. This combination, together with a deficit in the driver's attention and the conception that nothing is going to happen will drastically increase the risk of an accident in traffic.
This study tried analyzing how the forward collision warning (FCW) systems could be used in decreasing the number of rear-end crashes. According to a number of studies, such systems are supposed to make the driver alert in case of an approaching obstacle that he or she may not be seeing. The findings apply to both the combined visual and acoustic models, as well as the acoustic-only ones.
In a 2002 study by Lee et al., it was found that the FCW systems were also useful to attentive drivers because, even though they were aware of the obstacle in front, they were quicker to respond than usual.
Even if the majority of studies showed that FCW had a number of positives, it couldn't stop the accidents from happening. Making the driver aware of the situation simply wasn't enough. As such, we integrate the autonomous braking system into the conversation. It can actually stop an accident from happening by activating the brakes, even without the driver showing any intention of doing so, making it different from assisted braking, which only corrects the pressure the driver is putting on the pedal.
We have to also keep in mind that too much autonomous intervention is likely to throw the driver off or make him drive in a riskier manner. This type of driving is not what the system is accustomed to, so it also makes it less efficient.
In order to conclude whether a partially autonomous driver assistance system would, in fact, be useful in decreasing the number of rear-end crashes, we have to look at the scenarios in which such a crash would have a high chance of happening. The two factors that a driver is focused on are: the road itself and the other vehicles around them. As such, we can deduct that these are critical to this crash scenario.
This is where the practical part of the study happens. The researchers tested drivers who were instructed to follow the car in front, while also keeping them distracted with a secondary visual task and simulating critical events (although the critical events were not present during the whole trial because they wanted to see if the drivers would engage in other tasks or not when conditions were normal). The secondary task I mentioned earlier consisted of finding a specific circle in a pool of one hundred others on a touch screen located in the passenger seat. The drives had to pick on which side of the screen the targeted circle was on. They were told to only work on the task when they deemed it safe.
The study found that the environmental cues played a big role in the driver's expectations. When the car in front would break right in the beginning, the drivers would assume that such an event is highly probable in the future, so they paid a lot more attention to it, thus diminishing the number of accidents. Also, the study participants tended to be less attentive on straight patches of road, and more in intersections and curves.
Even though the auditory signal was not faster than the driver's perception of the lead car braking, the system definitely helped the reaction time, making the autonomous interference crucial in stopping the crash.
It was concluded that the majority of the problems encountered during the study could be easily corrected with a revised human-machine interface.
In the end, the drivers were better protected in the case of a crash when the

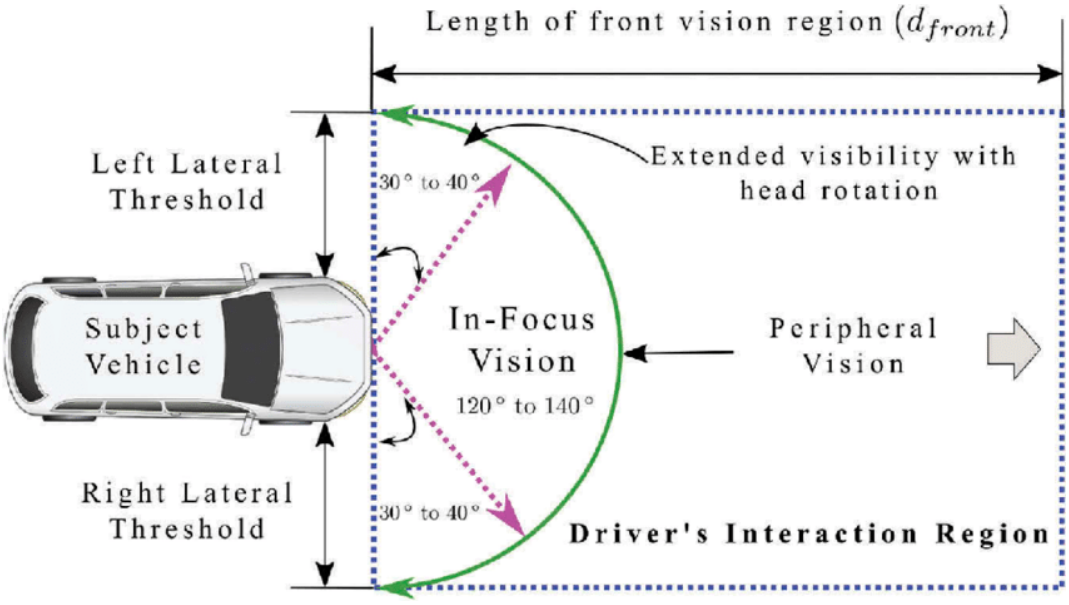

Figure 3.1: Interaction Region of a driver [2]

autonomous systems were on, making the small number of negatives worth it in the end.
This was an important finding for my thesis, which proved once again the importance of such a system, motivating that in the future development, it will be worth to also have a system connected to the brakes for much more safer situations.

I also have found this important schematic shown in Figure 3.1 of the interaction region of a driver, so I could base my collision warning radius[1] on it.

[1]collision warning radius = the area in which a detected traffic participant is considered an immediate danger and a warning will be sent

Based on another study [8] that I found where 45 drivers are tested whether an adaptive front collision warning system is more efficient than a regular one. Among these drivers they had two types:the non-aggressive drivers and the aggressive ones. The non-aggressive ones didn't show any preference for the adaptive one, however, the aggressive drivers preferred the adaptive front collision warning system due to it being not so irritating and stressful. The tests consisted in using a driving simulator, basically a real car with all it's driver controls and instruments fully functional that has a giant cylindrical screen around it with 2 speakers. On that screen is projected a simulated world and behind the car there was a smaller screen that can be seen in a rear pier glass pointing to it. The FPS (frames per second) was set to a consistent 60 Hz, the test system was an immovable-type (the car wasn't moving, the screens basically simulated the movement), combined with two electrical motors for simulating the steering wheel torque and brake pedal assistance. The Figure 3.2 represents how the simulator looked in the experiment.

Figure 3.2: Leeds Driving Simulator [8]

The collision warning system was based on the SDA (Stop Distance Algorithm)[13], which basically computes if the difference of stopping distances between the ego vehicle and the vehicle in front is smaller than a threshold, an acoustic warning will be sent to the driver of the ego vehicle, thus alerting him of a possible crash.
This system was also adapted (for the adaptive collision warning participants) by modifying the ego vehicle driver's reaction time every time a possible crash was avoided, to predict the next one better. The other system which wasn't adaptive had a fixed driver reaction of 1 second and a half.
Both types of drivers showed improvement in their reaction time before a collision was happening regardless of the system used (adaptive or non-adaptive), resulting in proving how effective a such system is for our world.

3.1 Functional requirements

- Detect traffic participants (cars, trucks, buses, motorcycles, bicycles and pedestrians);
- Predict current trajectory of the ego vehicle;
- Alert the driver of a possible crash;
- Have different settings to match the user's criteria;
- Have different vehicle types in which the program is used;
- Compute and display a FPS counter;
- Have a complex trained model to detect objects with high accuracy.

3.2 Non-functional requirements

- The program needs to perform fast enough in order to make it useful in intense traffic conditions;
- It needs to operate as long as the program is open (24/7 service time)
- As for the limits, it needs to handle as many traffic participants as the camera of the device can see;
- In terms of reliability, it needs to have the rate of error as low as possible;
- It needs to be easy to use, so that it can be used by any person that wishes to have it on his/her vehicle;
- It should handle only one user at that specific time on that device.

Chapter 4

Author Contribution

4.1 Implementation Manual

My solution was to create a Python program because I was very familiar with it and it works very efficiently with videos and frames. For that I used the Python version 3.9.13 [12], as it was the latest one available when I started working on the thesis, and a graphical user interface library called PySimpleGUI. I created a main window, in which I placed the 3 main buttons:

- START DETECTION
- TEST DETECTION
- EXIT

The first button is starting the real time video detection in a new window , the second button is used to test the current settings on a pre-recorded video, also in a new window. The last button as the name suggests, is used to close the program. Below these buttons, there are 3 more buttons which have have the role to adjust the detection parameters, for example, how far an object needs to be in order to trigger a warning. Currently, these 3 settings are the following:

- CITY ROAD setting, which has a smaller but wider collision warning radius[1] when a detected traffic participant enters it
- HIGHWAY setting, which has a bigger but narrower collision warning radius
- NORMAL ROAD setting, which is a more balanced solution between the other two settings

Now for the detection part, the graphical user interface "START DETECTION" button (also the "TEST DETECTION button does the same but with a different input) calls the "detection()" method in which Tensorflow version 2.8.0 [1] is used (it's an open source platform for developing, training and running models) along with two parameters:

- input, which is the video that is used (either the webcam or a recording)

[1]collision warning radius = the area in which a detected traffic participant is considered an immediate danger and a warning will be sent

- roadType, which is the current active setting for the type of road that the user is encountering

The first line of the detection method is calling "modelSetup()" in which I had to load the labels using a label map; the labels are basically tags used when the convolution network predicts an object, for example a '3', and with the help of labels I now know that it maps to a car object. In my case I used the SSD Mobilenet v1 COCO 2017 [9] frozen graph (model), and removed the categories that aren't a traffic participant and left the important ones:

- ID=1: "Person"
- ID=2: "Bicycle"
- ID=3: "Car"
- ID=4: "Motorcycle"
- ID=6: "Bus"
- ID=7: "Train"
- ID=8: "Truck"

The 5th one was removed because it was "Airplane" and it wasn't related to the problem.
First I created four parameters:

- MODEL_NAME (a string containing the name of the model used),
- PATH_TO_FDG (another string containing the "MODEL_NAME" and the name of the frozen detection graph pb file),
- PATH_TO_LABELS (the actual path to the labels)
- NUM_CLASSES (the number of classes used)

After that, I loaded the model in the memory by calling Tensorflow's method tf.Graph() along with tf.compat.v2.io.gfile.GFile() and as a parameter I had to send the path to the frozen detection graph.

Next we have a function call for the label map, from the label_map_util library, the load_labelmap() along with it's path to my labels as a parameter. Now I had to create a categories variable to store the ones created from the label map, and from this we got the category index, which is the mapping of the number and name shown above. The modelSetup() function will return the detection graph along with the category index to be used in the detection part. Now I have the initialized model, which is used to create a Tensorflow session. Inside the session the input video is processed frame by frame as the following work flow diagram shows in Figure 4.1 , with the following steps:

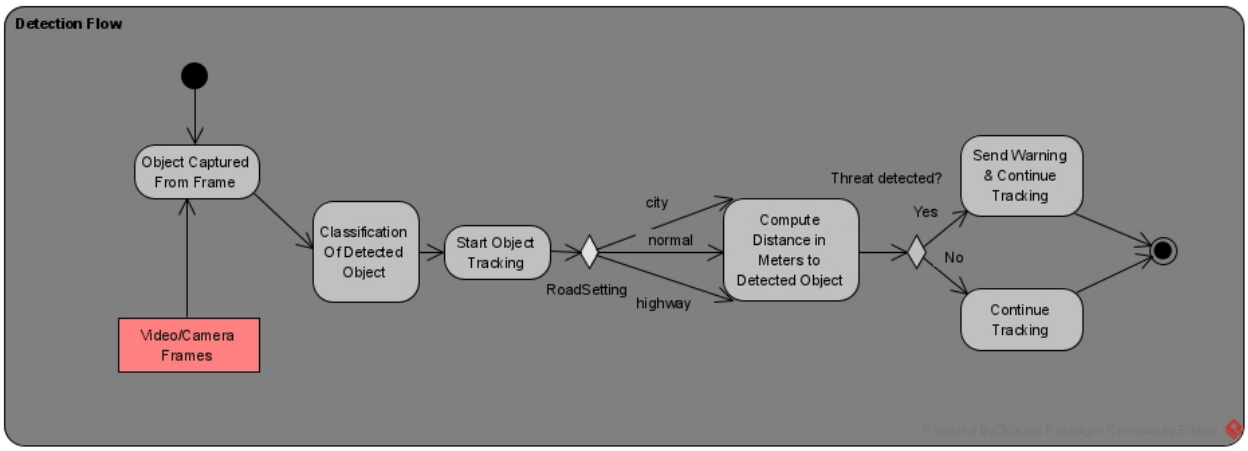

Figure 4.1: Work Flow diagram of the Detection Flow

- The frames are coming from the "CameraFrameGetter" class which uses another thread, in order to be processed faster;
- The CameraFrameGetter class looks like this:

```
from threading import Thread
import cv2


class CameraFrameGetter:
    def __init__(self, src=0):
        # initialize the camera stream
        # and read the first frame from input
        self.input = cv2.VideoCapture(src)
        (self.success, self.frame) = self.input.read()
        # flag for thread status
        self.stopped = False

    def start(self):
        # thread is created and started reading frames
        Thread(target=self.update, args=()).start()
        return self

    def update(self):
        # this keeps the thread alive
        while True:
            # if a flag is triggered, kill the thread
            if self.stopped or not self.success:
                return
            # continue reading from input
            (self.success, self.frame) = self.input.read()

    def read(self):
```

```
            # returns the last frame read
            return self.frame

        def stop(self):
            # changes the thread's status flag to stopped
            self.stopped = True
```

- First, the frame returned gets expanded dimensionally by 1 on the row side, because the model expects it to be that way;
- Next, a tensor[2] is created for the expanded frame (this will get called later together with the frame in the "run()" method of the session);
- Afterwards, a tensor is created for each of the following:
  - for the current frame
  - for detection boxes
  - for detection scores
  - for detection classes
  - for the number of detected objects
- The session is run using the above tensors as parameters and the detection starts;
- Next, the visualization process begins by calling vis_util.visualize_boxes_and_labels_on_image_array with the same parameters as the ones from above, but the tensors get "squeezed" using the NumPy[6] method "np.squeeze()" on each of them
- Now the object detection part is finished and the program proceeds with the creation of a thread used to call the "computeDistancesAndSendWarning()" along with the following arguments:
  - frame, which is the current frame;
  - tensorBoxes, which is the tensor for the detection boxes;
  - tensorClasses, for the detection classes;
  - tensorScores, for the detection score;
  - and the roadType, which is the current road setting;
- The newly created thread is started and joined with the main thread
- The "computeDistancesAndSendWarning()" is a very clever function which has the following instructions:
  - iterate through all the boxes;
  - checks if the detection score is bigger than 40

[2]a tensor is a multi-dimensional array which has a shape (how many rows and columns) and a uniform type (e.g. float32, float16, int32 etc.)

- if the case from above matches, a distance is computed using the box's width and it gets converted to approximate meters;
- then the middles of the X and Y box coordinates are saved in two new variables;
- for each detected traffic participant that meets the required score, the approximate distance is displayed above it using the OpenCV's[3] "cv2.putText()" method, middle X and Y are used as text position coordinates;
- afterwards, if the detected object is a vehicle and if it's inside the corresponding range of middle X specific for the selected road (an explanation of this concept is seen in the Figure 4.2) and the approximate distance is smaller than the threshold of the selected road;

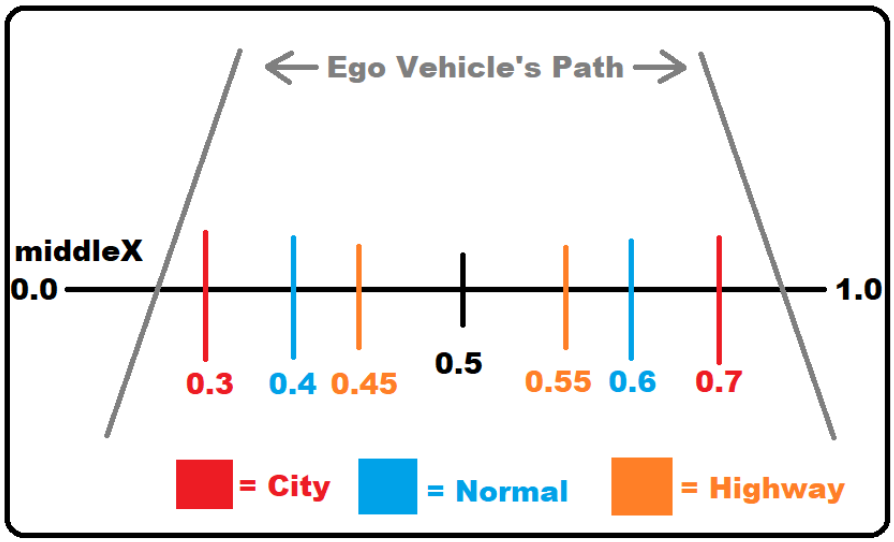

Figure 4.2: How the selected road setting influences the ego vehicle's predicted path.

- these two measurements create the collision warning radius, in which if a traffic participant is detected, a warning message is sent using the again "cv2.putText()" method;
- if the detected object is a person, the collision warning radius is larger and it is not influenced by the road type, because pedestrians are quite unpredictable and should be more protected by such systems; the Figure 4.3 shows this behaviour;

Figure 4.3: Example of pedestrian warnings, on normal road setting, the collision warning radius is not influenced by the road type

- After the warnings are sent, the current frame time is saved in order to compute the video's frames per second and display it on the top left side of the output window;
- The same algorithm is applied for the "TEST DETECTION" part, except the input is not processed in the "CameraFrameGetter" because it would have been too fast and a lot of frames would be skipped and lost.
- The detection works even when the light conditions are bad, so it can also be used at night, as seen in Figure 4.4

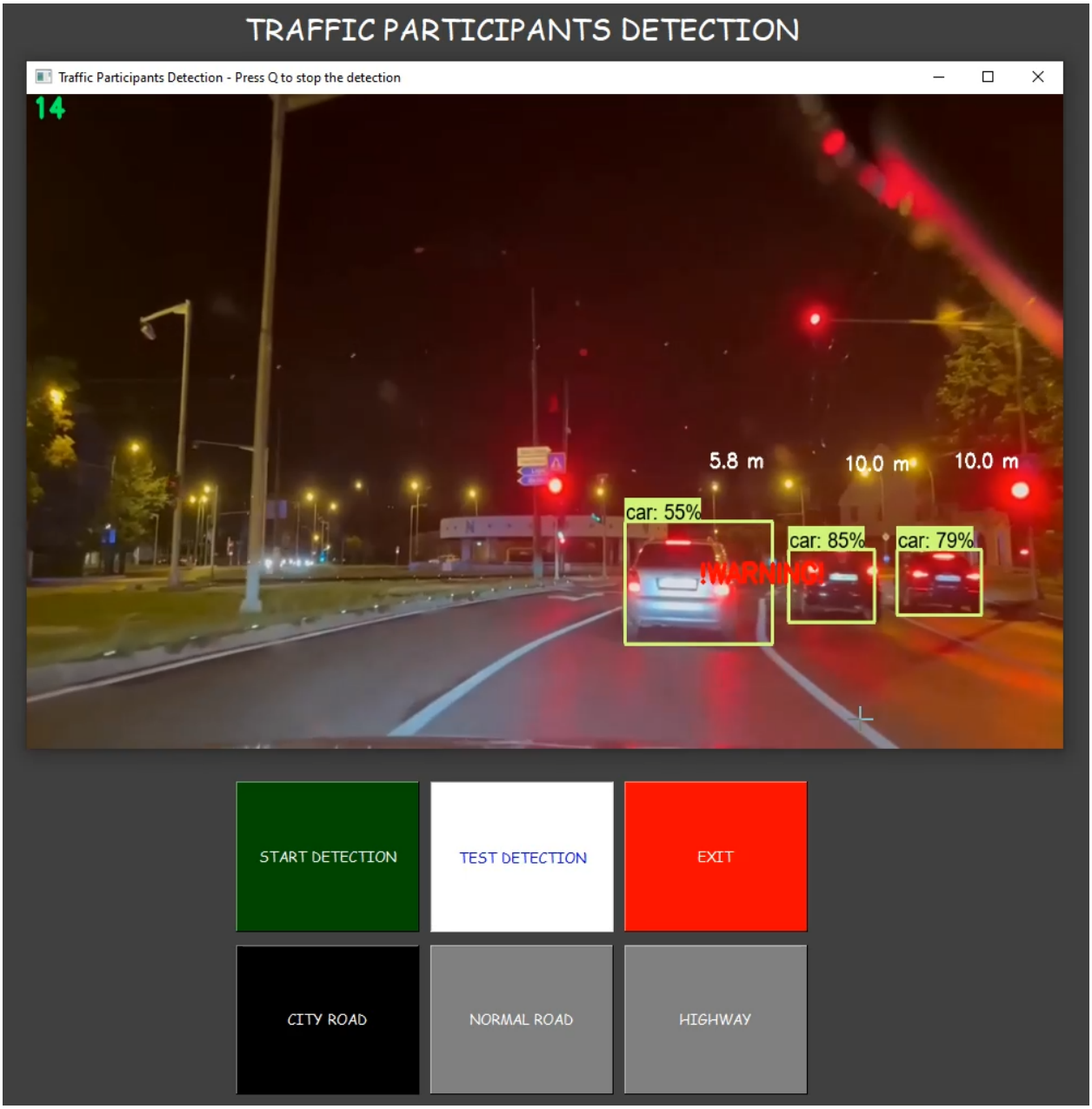

Figure 4.4: Example of program running with low light conditions in the city.

4.2 Application Architecture

All the use cases of the program can be seen in Figure 4.5. The three actors are: the user, the video camera and the recorded video (depending if the button pressed was "START DETECTION" or respectively "TEST DETECTION"). The user actor can also select the type of road, which directly influences how the warnings are sent. The last use case named "Exit" is used for closing the application.

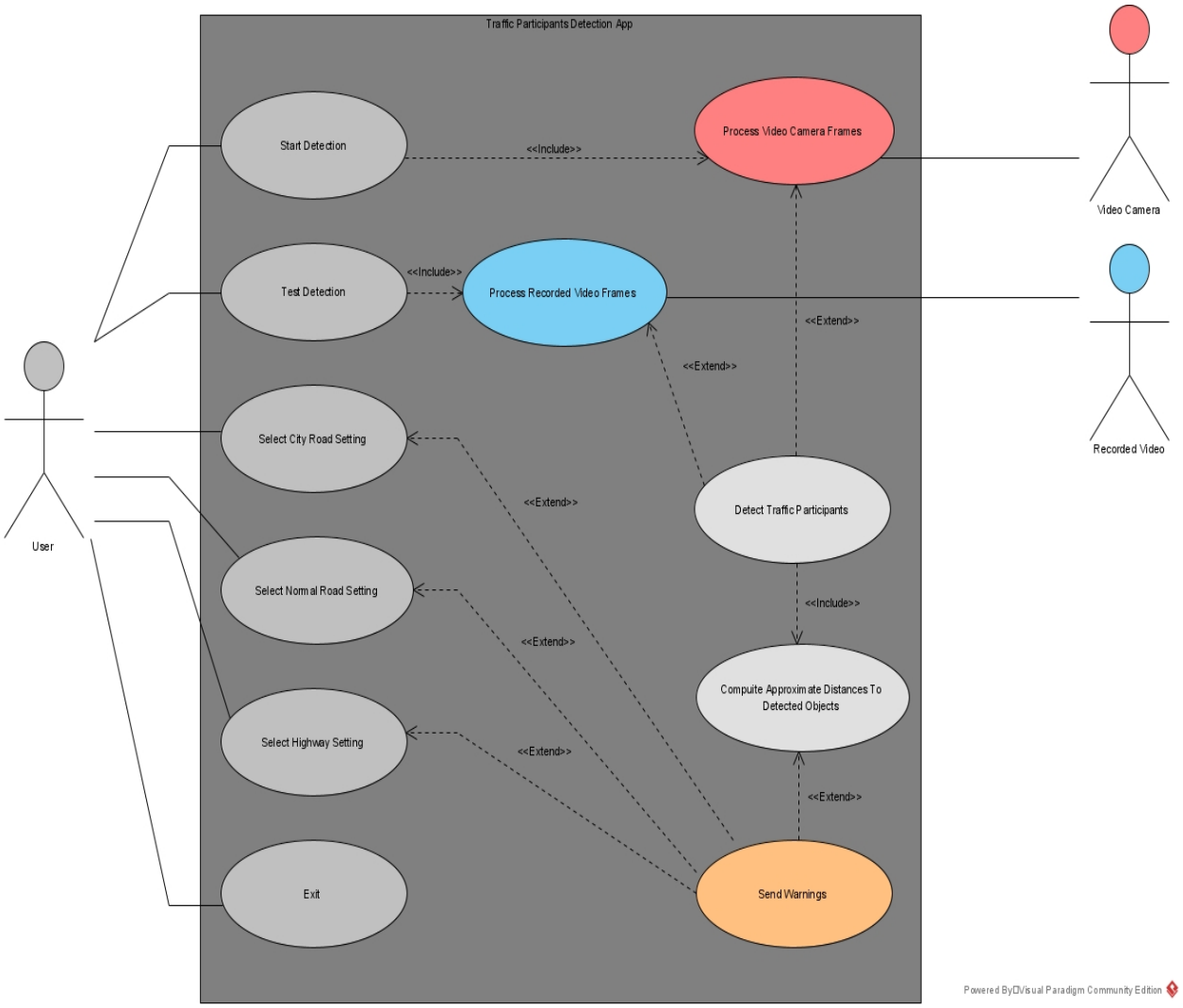

Figure 4.5: The use cases of the Traffic Participants Detection app.

The main program components can be seen in Figure 4.6, which are the main menu alongside with the two detection screens depending of the type of input. On the main menu screen we have six buttons, three are for starting, testing and exiting the application and the other three are for choosing the current road setting. On the detection screen, we can see the current frames per second in the upper left corner, the detected and classified traffic participants, the computed distances for the detected traffic participants and also any present warnings.

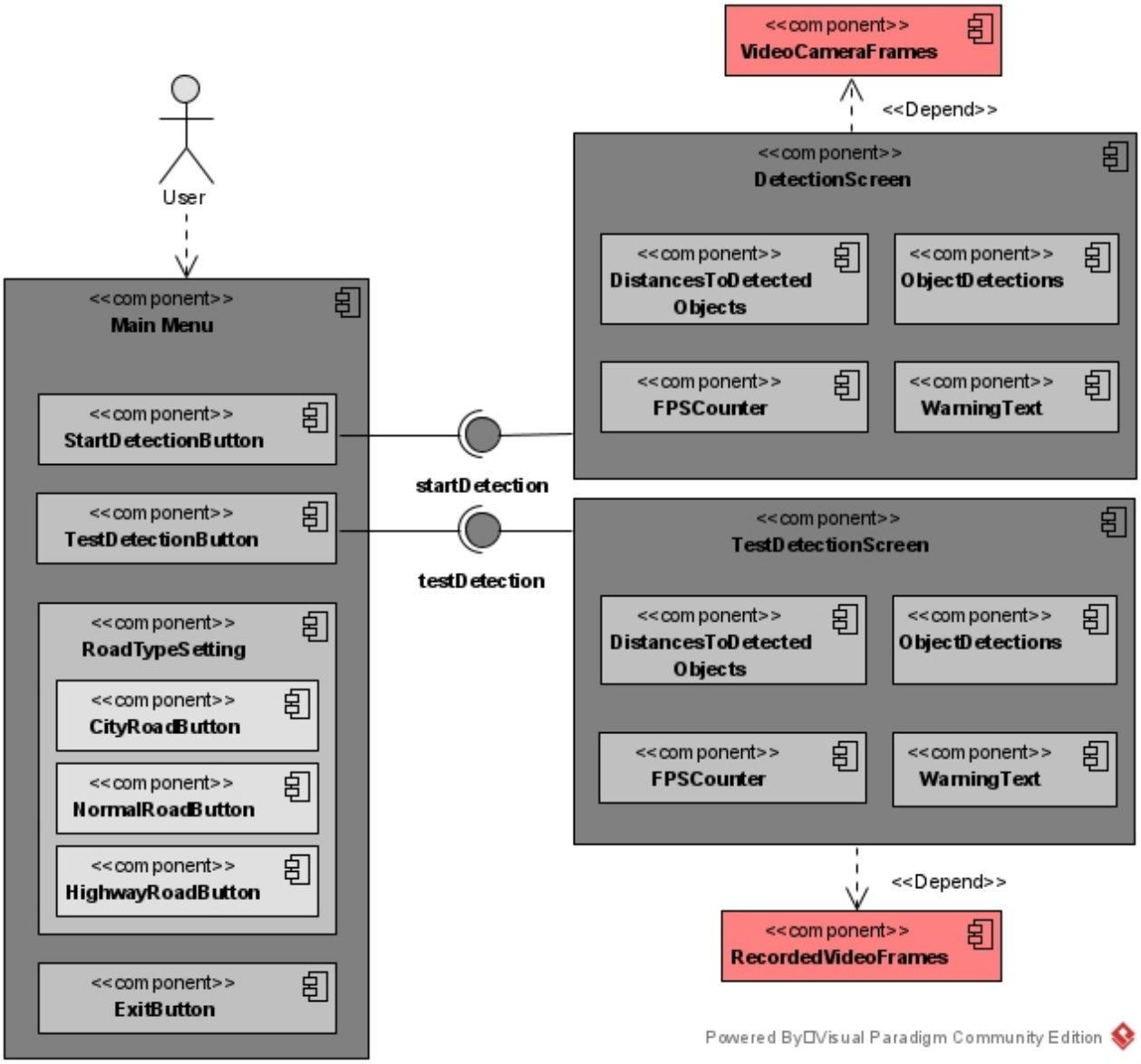

Figure 4.6: The use cases of the Traffic Participants Detection app.

4.3 User Manual

4.3.1 General Information

Traffic Participants Detection (TPD) is a Python application designed to assist drivers in predicting a front collision situation by detecting each traffic participant, regardless of it being a car, a motorcycle, a truck or a pedestrian. The purpose of it is also to equip older cars who do not have an ADAS (Advanced Driver Assistance System) in them with this important function. The system uses a Tensorflow model to detect the participants and afterwards it computes if it's in the ego vehicle's path or not, along with its approximate distance to collision.

4.3.2 System Overview

The system requires a laptop or a small computer with a capable CPU (a graphics card is recommended but not mandatory), a webcam (a HD webcam is recommended for better accuracy of detection) and a vehicle in which it can be placed. If the computer has a graphics card equipped, it will also need to have installed the NVIDIA GPU Computing Toolkit CUDA with version 11.2 or newer and the NVIDIA CUDNN driver version 8.1 or newer, so the Tensorflow will see and use the GPU instead of the CPU.

4.3.3 Getting Started

In order to install the TPD application, you will need to follow these steps in order:

- Install Python 3.9.13 from this link, scroll to the end of the page and select the Windows installer (64-bit) or Windows installer (32-bit) depending on your type of system:
  https://www.python.org/downloads/release/python-3913/
- While installing Python, make sure to check the "Add Python to environment variables" option
- After the installation has finished successfully, make sure you have the latest version of pip by using this command:
  python -m pip install –upgrade pip
- Clone the TPD GitHub repository:
  https://github.com/PaulCvasa/TPD.git
- Install all the required libraries for the application by opening a command prompt in the TPD folder and running this command:
  py -m pip install -r requirements.txt
- Download the Tensorflow API from the Google Drive shared folder: https://drive.google.com/file/d/1DUmediZ2HRg4sQrkxwJcq000byVsJqGV/view?usp=sharing
- Extract the Tensorflow folder in the same directory as the TPD.py script

- If the device running the program doesn't have a GPU, skip the next 10 steps, and comment these 2 lines from the TPD.py script:

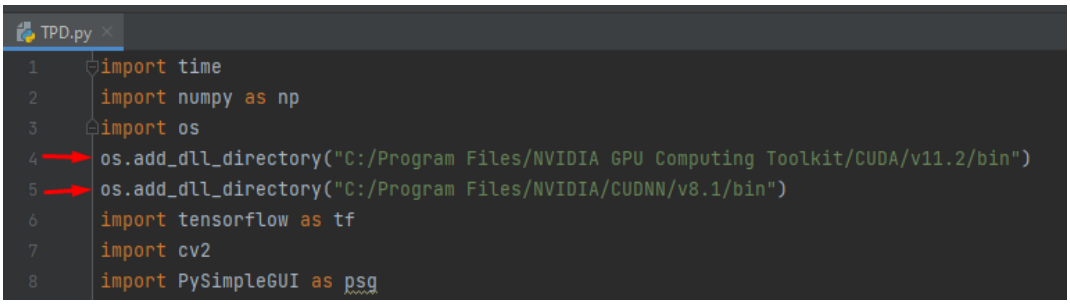

Figure 4.7: Lines that need to be commented if the device isn't equipped with a GPU

- If the device is equipped with a GPU, download the CUDA Toolkit 11.2 from here: https://developer.nvidia.com/cuda-11.2.0-download-archive
- Once you open the NVIDIA website link, select the Windows operating system, then the x86_64 architecture, the selected version should be 10 (the first one along Server 2019 and Server 2016) and the installer type exe (local). After that a download button will appear.
- Install the CUDA Toolkit in the following path: C:/Program Files/NVIDIA GPU Computing Toolkit
- After the setup finished successfully, add this to your system's environment variables PATH: C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v11.2/bin
- Next, Tensorflow also needs a NVIDIA cuDNN library of version v8.1.0 in order to be more optimized for deep neural networks and faster in detection. This library can be downloaded from here: https://developer.nvidia.com/rdp/cudnn-archive
- On that website, scroll down until you find the correct version ("Download cuDNN v8.1.0 (January 26th, 2021), for CUDA 11.0,11.1 and 11.2"), after selecting it, in the "Download cuDNN v8.1.0 (January 26th, 2021), for CUDA 11.0,11.1 and 11.2" section, find the one for Windows ("cuDNN Library for Windows (x86)")
- After clicking on that library, a login/register will be required for the "NVIDIA Developer Program Membership", then, a download will start.
- Next step would be to go in C:/Program Files and create a new folder called "NVIDIA". Inside that folder create another folder called "CUDNN" and inside it a new one called "v8.1".
- Unzip the "cuda" folder contents from the downloaded archive in: C:/Program Files/NVIDIA/CUDNN/v8.1

- Add the following folder to the system's environment variables PATH: C:/Program Files/NVIDIA/CUDNN/v8.1/bin
- Finally, the script TPD.py needs to be executed and the program will run with no errors.

4.3.4 Using The Software Application

After the program is executed, the main screen (Figure 4.8) will appear:

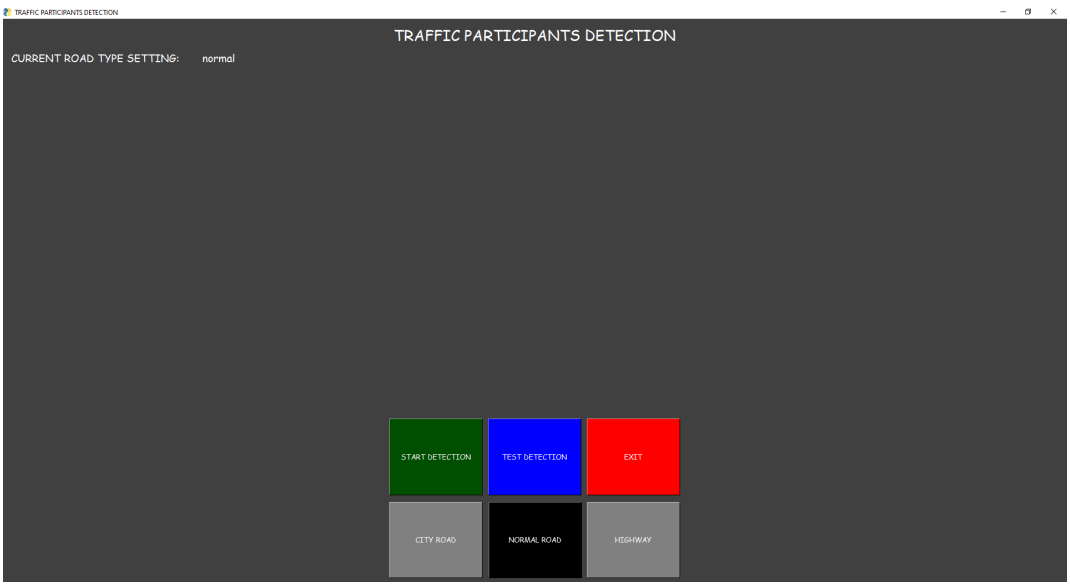

Figure 4.8: Main screen user interface

Thus the user will be presented with the following options shown in Figure 4.9:

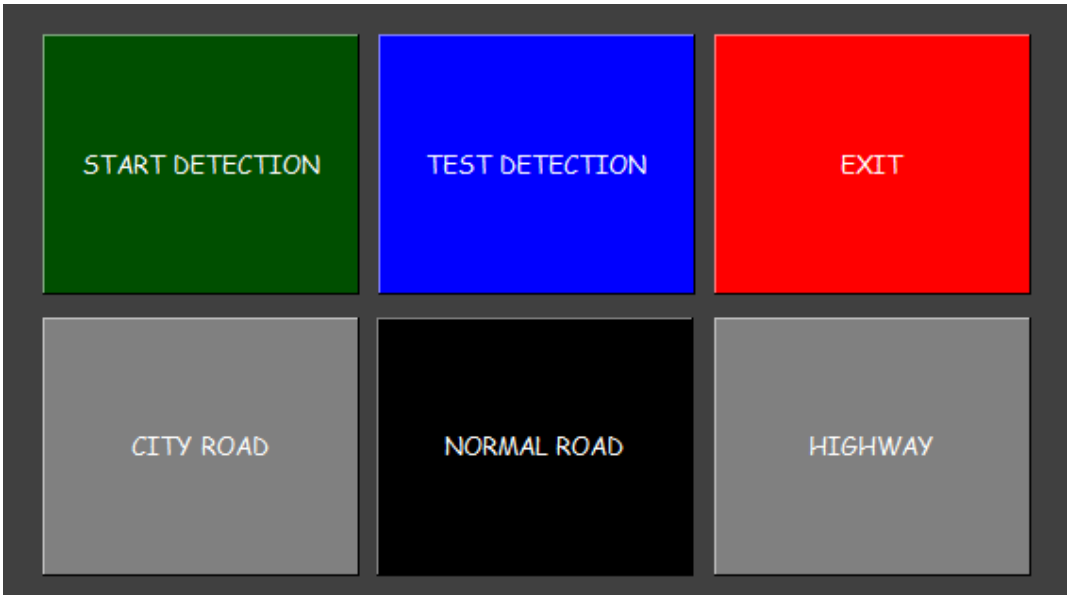

Figure 4.9: Main screen's buttons

- In order to start the application and use it in a vehicle, press the "START DETECTION" button; this will use the device's camera for video input and start the detection process in a new window.
- The next button is for testing purposes, so the video input will be coming from a pre-recorded traffic situation. To test it, the user should press the "TEST DETECTION" button and a new window will pop out, which can be seen in Figure 4.10.

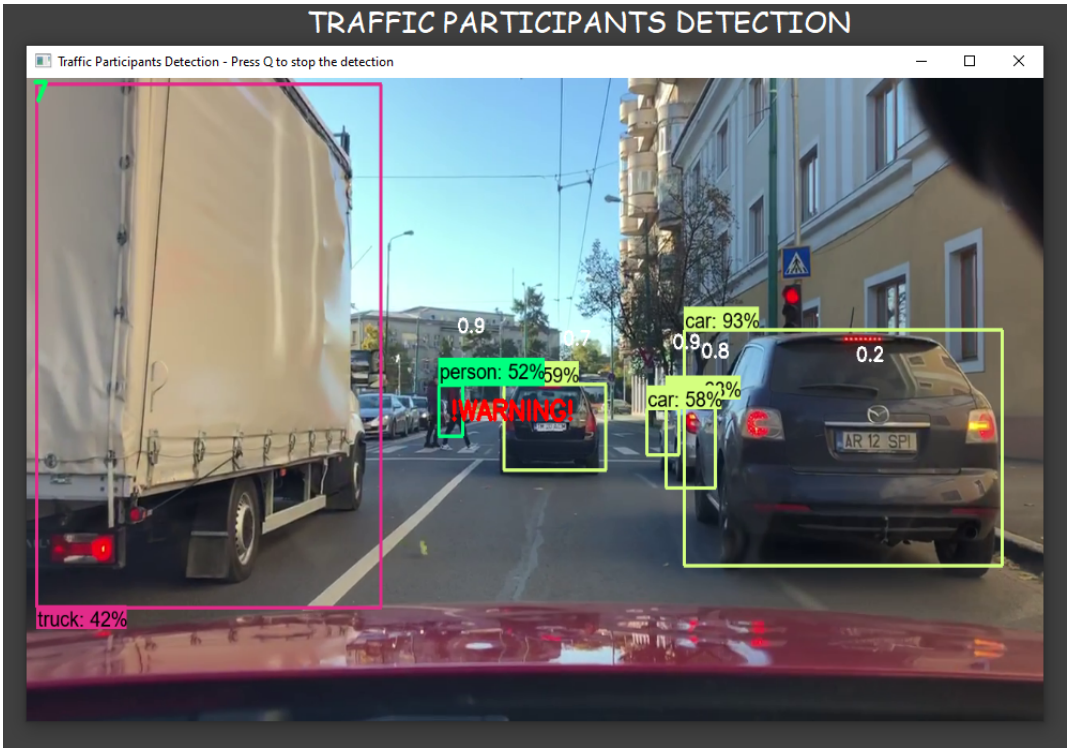

Figure 4.10: The testing video window

- The last button from this row is the "EXIT" button, pressing it will shutdown the application.
- On the next row, the user will have 3 available buttons, one for each desired road setting, the "CITY ROAD" button is used when the vehicle is inside a city, so the distance needed to send a warning will be shorter and the vehicle's collision path will be wider.
- The "HIGHWAY" button is used for highways where the traffic participants are moving considerably faster, the distance-to-warning is much more bigger and the vehicle's collision path is narrower.
- The "NORMAL ROAD" is generally used for roads outside the city but not fast enough as the highway setting, it's a balanced setting between the "CITY ROAD" and the "HIGHWAY" settings.
- By default, the road setting is set to normal.

In the upper left corner of the screen, the user is informed of which setting is applied at the current moment (that part of the screen is in Figure 4.11):

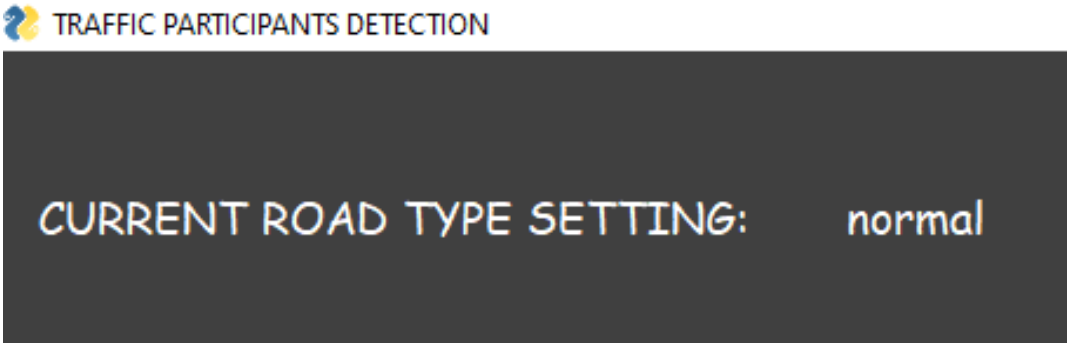

Figure 4.11: Current active setting display

4.3.5 Frequently Asked Questions (FAQ)

- How to close the detection window?
- - Press the Q key from the keyboard

- Is it possible to use the CPU instead of the GPU (after already doing the setup for GPU) ?
- - Yes, just comment the lines 7 and 8 (before the "import tensorflow as tf") containing the paths to the CUDA and CUDNN directory and Tensorflow won't detect your GPU and use the CPU instead.

- How can I improve my frame rate?
- - Use a better device, or change your camera's settings to have a lower quality video output.

4.4 System Requirements

- Processor Model: Intel Core i5/i7 6-th Generation or newer/AMD Ryzen 5/7 1700 or newer
- Processor Speed: 3 GHz or more
- RAM: 2 GB or more
- OS: Windows 7 or newer
- Video Card (optional): NVIDIA GEFORCE GTX 970 or newer/AMD Radeon R9 390 or newer
- DISK SPACE: 5 GB
- Any External Webcam

4.5 Software Requirements

- Python version 3.9.13

  Python[12] is an OOP(Object Oriented Programming) language in which this program was written combined with using some libraries from it in the aid of object detection, arrays processing, adding dll's directories to executable path, working with frames and creating a graphical user interface.

- Tensorflow version 2.8.0

  Tensorflow[1] is a library from Python for working fast with graphs made out of nodes represented by a math symbol. The nodes are connected by a tensor(a multidimensional array) are used to run a model and detect, in this specific case, the traffic participants.

- OpenCV

  The open-sourced computer vision library (also known as OpenCV) is a Python library used to read and write frames captured from a video, process frames, display the resulted frames with detections from Tensorflow and also add text on top of them.

- NumPy

  NumPy[6] was useful for working with arrays (as frames are made up of multiple arrays) and have different transformations of them (e.g. squeeze and expand dimensions).

- PySimpleGUI

  PySimpleGUI is another Python library that wraps tkinter (another library for graphical user interfaces but more complex) in a simple to learn, very fast and highly customizable graphical user interface.

- PyCharm

  PyCharm[4] is a popular Python IDE (Integrated Development Environment) developed by JetBrains with the capability of creating a virtual Python environment for the developed application. It also helps by having an intelligent code editor, easy refactoring, quick suggestions to solve warnings and errors, integrated version control systems and import assistance for any missing libraries. This was mainly used for developing the Traffic participants Detection program.

- NVIDIA CUDA Toolkit version 11.2 or newer (optional)

  NVIDIA CUDA Toolkit[11] is made out of different libraries used to develop applications that are using the graphics processing unit, by accelerating it, having a separate compiler and having the CUDA runtime. This is needed for the Tensorflow to use and see the available GPU-s.

- NVIDIA cuDNN version 8.1.0 or newer (optional)

  The CUDA[11] Deep Neural Network (cuDNN) is another library used by Tensorflow to take advantage of the availability of a graphics processing unit and make the computations faster.

Chapter 5

Conclusions and Future Work

As the traffic becomes more and more crowded and dangerous for everyone, it's very important to create a safer environment using the benefits of the newly developed ADAS (Advanced Driver Assistance Systems) technologies. Especially the front collision warning, which should be equipped to any type of vehicle, even it being a tram, a motorcycle or a bicycle; it serves the same purpose of preventing a crash caused by a driver's moment of distraction.
It is clear that such a system it is very useful to help the driver avoid the rear-end crashes. If every traffic participant will have a front collision warning device equipped and running, it will not only aid in saving lives of pedestrians and even drivers at high speeds, but also prevent losing money in damages and repairs needed for the bumpers and parts destroyed in such accidents. The application is for everyone's benefit, including the profit of any insurance company.

As for future updates, work will be done on the following topics:

- Improve performance of frames per second;
- Use/create a new model for more accurate detection;
- Create a script that will do the setup for the program automatically;
- Add road edge detection;
- Add lane departure warning;
- Add traffic signs and traffic lights recognition;
- The option to automatically detect the type of road.

Bibliography

[1] Martín Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Rushikesh Amrutsamanvar, Bharathiraja Muthurajan, and L. Vanajakshi. Extraction and analysis of microscopic traffic data in disordered heterogeneous traffic conditions. *Transportation Letters*, 13:1–20, 11 2019.

[3] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[4] Sergey Dmitriev and Max Shafirov. Pycharm, 2010.

[5] INFOCOM Ltd Eduard Trotsenko. Ugv driver assistant, 2018.

[6] Charles R. Harris et al. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[7] Razvan Itu. Drive assist, 2014.

[8] A. Hamish Jamson, Frank C.H. Lai, and Oliver M.J. Carsten. Potential benefits of an adaptive forward collision warning system. *Transportation Research Part C: Emerging Technologies*, 16(4):471–484, 2008.

[9] Tsung-Yi et al. Lin. Microsoft coco: Common objects in context, 2014.

[10] Elke Muhrer, Klaus Reinprecht, and Mark Vollrath. Driving with a partially autonomous forward collision warning system: How do drivers react? *Human Factors*, 54(5):698–708, 2012. PMID: 23156616.

[11] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[12] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[13] Terry B. Wilson, Walker Butler, Dan V. McGehee, and Tom A. Dingus. Forward-looking collision warning system performance guidelines. *SAE Transactions*, 106:701–725, 1997.