INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Simulator for Driving Performance Monitoring

DIOGO ALEXANDRE RODRIGUES RAIMUNDO

Graduate

Dissertion submitted for the Degree of Master
in Engenharia Informática e de Computadores

Supervisors : André Lourenço, PhD
Arnaldo Abrantes, PhD

Jury:

President: Fernando Manuel Gomes de Sousa, PhD

Vowels: Rui Manuel Feliciano de Jesus, PhD
André Ribeiro Lourenço, PhD

DECEMBER, 2017

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Simulator for Driving Performance Monitoring

DIOGO ALEXANDRE RODRIGUES RAIMUNDO

Graduate

Dissertion submitted for the Degree of Master
in Engenharia Informática e de Computadores

Supervisors : André Lourenço, PhD
Arnaldo Abrantes, PhD

Jury:

President: Fernando Manuel Gomes de Sousa, PhD

Vowels: Rui Manuel Feliciano de Jesus, PhD
André Ribeiro Lourenço, PhD

DECEMBER, 2017

Acknowledgments

I would like to thank my supervisors André Lourenço and Arnaldo Abrantes for all the support, guidance, knowledge and patience given throughout this project. By always being available to advise me, they motivated me to work harder, better, and helped me focus on the ultimate goal.

A special thanks to the Engineers Carlos Carreiras, Ana Priscila Alves and all the *CardioID* team for helping me integrate their *CardioWheel* system into my project and for providing me tools to work with. I would also like to thank Professor Pedro Mendes Jorge for helping me prepare the setup used in the presentation of this project on the *ISELAlive* event.

Last but not least, a very special thanks to my family, girlfriend and friends for the support, motivation, strength and love that they always have given to me and that made the person I am today. For my parents and girlfriend: thank you for always support me when I has felling low, for listening to me talking "gibberish" and for always making sure I knew that they were there. I couldn't be more thankful for having you in my life.

Abstract

Driver and driving behaviour monitoring are a reality on new cars and on old fleets, driven by increased awareness about safety and environmental concerns. The numbers of car crashes are decreasing but still represent a huge concern, leading to the renewed interest on driving research topic.

This thesis is centred on this theme and aims develop a driving simulator and modular software architecture that enables the variables related with the simulated vehicle, and that at the same time, acquires physiological, in particular the electrocardiogram (ECG), and behavioural variables from the driver.

These data are registered aiming developing algorithms for drowsiness and fatigue, and also other related effects. The developed system is based on a message management and distribution module that uses the Publish/Subscribe pattern and the Open Sound Control communication protocol. All modules, including the simulator and the ECG module, communicate through this central module. The simulator is developed on the Unity platform and has some elements of artificial intelligence to control other cars present in the simulation.

The results show that the proposed architecture is adequate for the proposed task, and can be used for drowsiness and fatigue studies. Preliminary results show that there are differences on driver variables (steering wheel movement) on drowsiness driving. More test are necessary to correlate with physiological variables.

Keywords: driving simulator; high fatigue driving; driving under the influence of alcohol; unity; boids; publish-subscribe; open sound control

Resumo

A monitorização do condutor e do seu comportamento durante a condução é, cada vez mais, uma realidade, sendo impulsionada por uma maior consciencialização sobre segurança e preocupações ambientais. O número de acidentes de carro, apesar de estar em diminuição, ainda representa uma grande preocupação, o que eleva o interesse em realizar estudos sobre o tópico. Esta tese é centrada nesse tema e visa desenvolver um simulador de condução e uma arquitetura de software modular que permita a aquisição de variáveis referentes ao veículo simulado e, ao mesmo tempo, sinais fisiológicos, em particular o eletrocardiograma (ECG) e descrições comportamentais do condutor.

Esses dados são registados com o objetivo de desenvolver algoritmos de deteção de sonolência e fadiga, além de outros efeitos relacionados. O sistema desenvolvido tem como núcleo um módulo de gestão e distribuição de mensagens que usa o padrão de *Publish/Subscribe* e o protocolo de comunicação *Open Sound Control* (OSC). Todos os módulos, incluindo o simulador e o módulo ECG, comunicam através deste módulo central. O simulador é desenvolvido na plataforma *Unity* e, utilizando técnicas de inteligência artificial, inclui outros carros presentes na simulação. Os resultados à arquitetura desenvolvida mostram que esta é adequada para a tarefa proposta e pode ser usada para estudos de sonolência e fadiga. Resultados preliminares mostram possíveis diferenças nos valores do condutor (movimento do volante) quando em condução sob sonolência. O mesmo não acontece com os dados fisiológicos devido à necessidade de mais dados e processamento dos mesmos.

Palavras-chave: simulador de condução; unity; boids; publish-subscribe; open sound control protocol; condução sob fadiga; condução sob o efeito de álcool

Contents

List of Figures

List of Tables

Listing

1

Introduction

Within the last century, the usage of simulators have been progressively included in the human live. As many of the technology used today, it began with war necessities, being later applied to public health, industry safety, comfort and many others.

Long before the computer invention, war generals used maps and boards to plan and simulate the enemy actions. On the First World War, there was too much pilots to properly train within the given time available, which led to a larger number of crashes and deaths. To solve this problem, engineers created a mechanical plane simulator that allowed to safely train more pilots simultaneously.

Initially, in the 1950s, computers were very large, slow machines, and until the 1970s, building a simulator was an expensive task that required highly specialized technicians. Later on the same decade, simulators were already used to solve security and optimization issues on car production industry and, by the 1990s, the reduction of equipment prices, the increase of its performance and the simplification of the techniques used, led to the diffusion of simulation as a efficient approach to many applications [11].

As mentioned before, the simulation of vehicles started as tools to train military pilots without the safety hazards but, with the digital revolution, they are now used in a wide range of applications, including driving training and research. This last usage of a driving simulators can take advantage of all the data used to model and iterate the system, that would otherwise require too much technology

to acquire in a real car.

Of course that, although the usage of driving simulators (laboratory tests) provides a controlled environment to test the subjects, they can still be complex and monetarily expensive to develop for a single experience [12]. This cost and complexity is reduced in the long term by developing the simulators as generic as possible and with maximum scalability, allowing to be reused by other researchers.

Another problem with laboratory tests is that the results may not represent true situations due to the controlled environment, described before as an advantage. In addition, the driver is aware that he is being evaluation and may try to alter is behaviour, even in a involuntary manner.

In theory, road tests (monitored driving) are less costly and may result in better and more objective measures, but with repeated experiments and eventual necessary modifications to the car, the costs start to add up. The solution can also be too complex to replicate or require the interaction with other people, which may not be acceptable accordingly to the study theme.

A more simplistic way to get data related to driving is through assessment based on questionnaires. This approach as the problem of being subjective and may not be very reliable due to generally not being verifiable by other means [58]. Sometimes is even required higher-order cognitive process (e.g. situation weighting, prediction, interpretation) from the test subject, which makes the data on the report not as raw as it should be.

1.1 Research Simulation Experiment

From all the driving hazards that can be addressed in a simulator-based research, this thesis will focus on fatigue and alcohol consumption, since their are two of the most common causes of deadly car crashes.

It is estimated that in the United States of America, between 2005 and 2007, 94% of the motor vehicle crashes occur due to human error [72]. In the same study it is also shown that 41% of these crashes are assigned to recognition error, which include lack of attention, internal and external distractions, and inadequate driver surveillance. Although only 7% of the crashes are directly assigned to falling asleep on the wheel, all the reasons mentioned on Table 1.1 can be caused by the driver having high levels of fatigue or sleepiness, which makes it one of the main causes of road accidents [64].

Table 1.1: Human-related Crashes (94% of the motor vehicle crashes estimation)

Crash Reason	*Percentage*
Recognition Error (e.g. lack of attention)	41%
Decision Error (e.g. inappropriate speed given the conditions)	33%
Performance Error (e.g. overcompensation)	11%
Non-Performance Error (mostly falling asleep)	7%
Other	8%

It is important to mention that, although the concepts of fatigue and sleepiness (or drowsiness) are similar, they are not synonyms. While "fatigue" describes the decay of attention level due to a demanding and/or repetitive process, "sleepiness" it is an individual's difficulty in staying awake [56] and it is considered to be the first stage of NREM (i.e. non-rapid eye movement sleep), which is the second stage of sleep [64]. Fatigue is considered by some researchers as the transient period between being awake and sleeping while sleepiness is directly related to biological sleep cycles that follow a daily rhythm.

It is also important to know that researchers [9] have compile a list of characteristics associated to drowsy driving crashes, such as the time when it occurs, the type of road, and number of people in the vehicle. The most common scenario is while the driver is alone, on the highway, from midnight to 7AM or from 2PM to 4PM. This selection characteristics may exclude some accidents caused by drowsiness that fail them, meaning that statistics using these criteria may be more devastating.

An article from 1976 [43] validated that as the driving time increased, the driver's reaction time became longer and the distance of object detection shorter. This conclusion can be questionable as it could be the driver compensating for his increased fatigue (e.g. choosing to slowdown the vehicle to avoid collisions). Another article [23] also reported that, as driving time increased, so did lane drifting and unnecessary speed variations, as well the eye closures duration.

However, a fatigued driver has the tendency to avoid effort [14], which may increase the danger (e.g. reduce speed when approaching a blind corner). This wish to simplify the action of driving a vehicle might also make the driver increase his safety margins and decrease speed, which "allows" him to operate at a lower level of vigilance. This relaxation of the driver can result in an accident if a more abrupt reaction is required.

In addition to driving with high levels of fatigue, another major reason of vehicle crashes is driving under the influence of alcohol or psychotropic substances. According to the USA CDC[1] [15], in 2015 29% of all road traffic fatalities were caused by driving under the influence. In Portugal, according to CEDIS[2] of the Law School of Universidade Nova de Lisboa [47], 33% of the fatalities in motor vehicle accidents had a blood alcohol content $\geqslant$0.5g/L. Even more alarming is the fact that $^2/_3$ of these victims had a blood alcohol content $\geqslant$1.2g/L which is considered a crime.

In 1958, a study from Manchester [16] showed a correlation between the space needed by an experienced driver to perform a manoeuvre, and its blood alcohol level. The same study also showed that intoxicated drivers were more confident in doing a difficult manoeuvre than they would had if they were sober. This are known effects of alcohol consumption: inaccurate assessment of risks, reduced awareness and changed arousal levels [26].

As reported by an article from 1997 [17], 17 hours of sleep deprivation impair performance to an extent equivalent to 0.5g/L of blood alcohol concentration and after 24 hours, the performance decreases to an equivalent of 1g/L. These similar impairments due to fatigue and alcohol intoxication can support policies aimed to reduce these situations. Another approach to reducing these road hazards is the development of systems or tools capable of analysing the driver and his surroundings in order to alert him or others that may support him (e.g. the police). To develop these systems it is important to have data related to driving both fatigued or intoxicated, which can be recorded through self-report, laboratory or road tests, as mentioned before [12].

1.2 Thesis Goals and Outline

This dissertation aims to develop a driving simulator supported by a modular and scalable communication architecture that would allow distribute and add modules throughout different machines and is part of a bigger project named SIMON (i.e. Driving Simulator: Fatigue and Sleepiness Monitoring).

One of the requirements of this project was the ability to decompose it into individual and independent modules that could be hosted by different machines.

[1] Center for Disease Control and Prevention
[2] Centro de Invertigação e Desenvolvimento Sobre Direito e Sociedade

The implementation of this concept, illustrated on figure 1.1, would need a flexible network communication architecture to support it, which that will also be developed on this thesis.

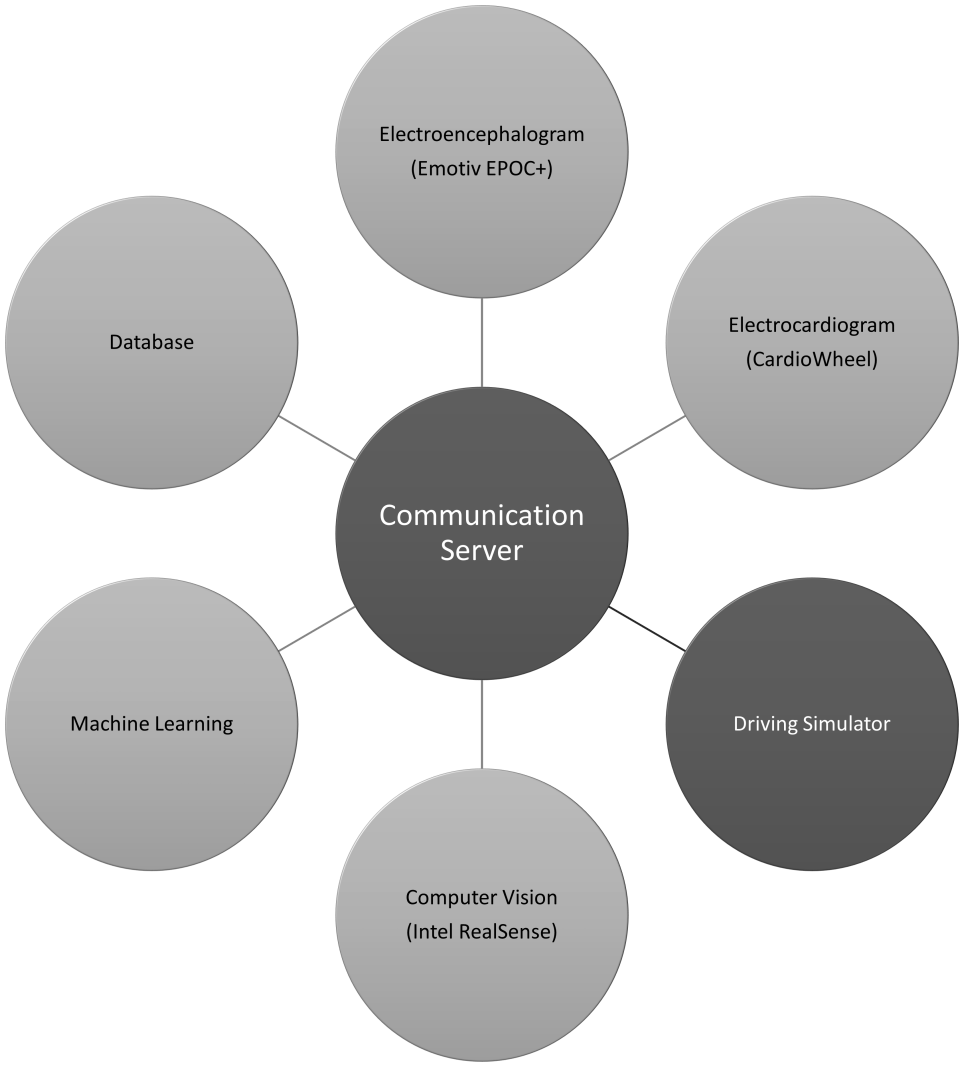


Figure 1.1: SIMON modules developed on the thesis (in black)

SIMON is an IPL[3] project that will later be used by partnered associations such as PRP[4], *CardioID Technologies* (developer of the *CardioWheel* system), and *ESTeSL*[5]. The system developed under the SIMON project will be open to use by ISEL[6] students. It is also expected to be used in the research of natural interfaces (e.g. gesture-based interfaces) by comparing the driver's performance while using different types of human-machine interaction systems.

[3] *Instituto Politécnico de Lisboa*
[4] *Prevenção Rodoviária Portuguesa*
[5] *Escola Superior de Tecnologia da Saúde de Lisboa*
[6] *Instituto Superior de Engenharia de Lisboa*

To test the solution developed, the data acquired will be used to detect the driver's level of fatigue, drowsiness, and alcohol intoxication, while also being displayed to the user on the panel of the simulated vehicle. Since sleepiness (drowsiness), alcohol and psychotropic substances are big "threats" to the good performance of drivers, there is a need to acquire and analyse data with multiple driving approaches and with drivers in several physical and emotional conditions.

It would not be safe or ethically right to test the driving performance of a tired or drunk subject on a real road, as it could result in physical injury to him or others. To avoid this risk, the main component of this project is the development of a driving simulator that collects the described measures while driving in multiple scenarios.

Among all the data that can be acquired regarding the drowsiness state of a driver, some studies [64] highlight the use of electrocardiogram (ECG) and electroencephalogram (EEG) [10, 42] as the most accurate approach to detect fatigue. Other measures that may be used are [64]:

- **Behavioural measures** such as head pose, yawning, eye closure or even eye blinking;
- **Performance measures** such as decrease of micro-movements on the steering wheel or deviation from lane position;
- **Subject self-measures** using the Karolinska sleepiness scale (KSS).

Since all the mentioned measures have advantages and disadvantages, this system should combine the strengths of all the collected measures, making it a more efficient hybrid to detect drowsiness.

This thesis will be organized as follows:

- Chapter 2 (State Of Art) - mentions some commercial and non-commercial driving simulators (both research- and entertainment-wise) and projects or researches develop in the area of fatigue detection;
- Chapter 3 (System Overview) - introduces and compares some software paradigms, patterns and protocols that could be used to develop the desired system;

- Chapter 4 (Architecture) - this chapter focus on describing all decisions made throughout the design and implementation phases of the core module that would to integrate all the other modules;
- Chapter 5 (Driving Simulator) - focus on describing all the details develop in the *Unity* engine framework, with special emphasis on the implementation of *Boids* and the integration with the *OSCD* system;
- Chapter 6 (Experimental Evaluation) - shows and describe some of the results obtain and uses them to describe how the system is working as expected;
- Chapter 7 (Conclusions and Future Work) - draws some conclusions using the results obtained and exposes some aspects to be addressed at future work.

2

State Of The Art

For many years researchers have been studying the effect of drowsiness and fatigue while driving on open road. From the simplest simulated environment to real cars fully equipped with technology that can acquire road events, car parameters, and driver's biosignals, numerous studies have been made to collect fatigue data and to develop solutions that could reduce this everyday problem.

2.1 Driving Simulators

First, it is important to define that a driving simulator is a system where the driver uses controls such as steering wheel, gear shift, and gas and brake pedals. The driver's actions are used by the system to calculate the position of a simulated vehicle and the view from the virtual driver's POV[1] is rendered and showed to the real driver in order to give him visual feedback. Driving simulators can bring numerous advantages to experiments and studies since they are used to evaluate the design of vehicles or to train drivers to operate expensive machines, such as tanks or trains. These simulators also have the advantage of be enable the user to repeatedly experience a situation that may occur rarely or is too dangerous to do in reality.

There are numerous articles and studies on fatigue detection and its effect while

[1] Point Of View

driving a vehicle. From the three approaches to driving analysis mentioned before, most of the reviewed studies choose to acquire the driver's data using a simulator. One of the studies [40] classifies driving simulators into three categories: low-level, mid-level and high-level. This classification is also used by NADS[2] from the University of Iowa, USA.

Simulators with one or more monitors, a realistic cockpit, a steering wheel, a gear box, and a set of 3 pedals are considered low-level structure (fig. 2.1(a)). If it uses advanced imaging techniques (e.g. large projection screen), a realistic car and even a simple motion base, it is classified as a mid-level, or fixed-base driving simulator (fig. 2.1(c) & 2.1(d)). A high-level simulation structure (fig. 2.1(b)), or motion-based simulator, should provide a 360º view and an advance moving base.

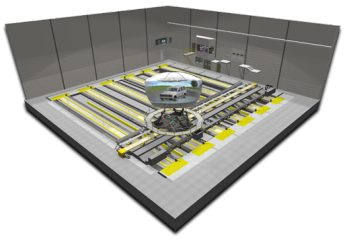
But it is of great importance to validate if a simulated experiment can be mapped to reality keeping the same parameters and results. In other words, this validity refers to the degree of reality-like behaviour under the same circumstances [40]. This means that any use of driving simulators should be followed by an evaluation of its validity, in order to give the experiments some credibility in the real world. Researchers mentioned multiple validations that can be done to a driving simulator, being one of them the *Face validity*. This validation refers to how realistic is an experimental environment to a given subject, which may not directly affect the experiment but could change how motivated is the subject that can directly affect the experiment.

A study from IBV[3] [63] used a low-level simulator to induce and measure drowsiness in a small sleep-deprived group of subjects. The experiments were done in a room with controlled light, sound and temperature, using infrared and thermographic cameras, pressure sensors on the driver's seat and a biomedical monitor to record his EEG, EOG and ECG signals, as well as his respiration and pulsoximetry (more on this topic in section 2.2.4). As motivation, the test subjects were told that they would receive an extra monetary reward if they could remain awake the entire experiment, which consisted of driving at night on a highway with smooth curves and low traffic for 1 hour and 45 minutes.

The group test subjects was divided in three phases classified by a control signal (table 2.1). In fact, none of the subjects was placed in phase 0, with 80% placed in phase I and the other 20% in phase II. This study reported a good performance

[2]The National Advanced Driving Simulator
[3]Instituto de Biomecânica de Valencia

(a) NADS MiniSim [70]



(b) NADS-1 [68]

(c) NADS-2 [69]

(d) Stanford University iMotions simulator [34]

Figure 2.1: Examples of low, mid, and high-level driving simulators

while discriminating the phases 0 and II but not the phase I. It also showed that the driving performance is a reliable measure to detect high risk situations but should always be complemented by another variable, such as PERCLOS[4] (section 2.2.2).

Table 2.1: Parameters used to define the control signal per phase

Variable	*Phase 0 (attentive)*	*Phase I (fatigued)*	*Phase II (drowsy)*
Behaviour	• High level of activity; • Fast reactions to road events; • Good lateral and longitudinal control.	• Slower reactions; • Yawns and large body movements; • Driving errors; • Loss of facial expressivity.	• Fall of attention to the road; • Departures of the lane.
EEG	• Lack of θ-waves[5]; • Regular patterns of α-waves[6] with closed eyes.	• Small ratio of θ-waves; • Regular patterns of α-waves with closed eyes.	• High ratio of θ-waves; • Loss of a regular pattern of α-waves with closed eyes.
PERCLOS	• Small PERCLOS; • Low and fast blinking.	• PERCLOS increase; • More frequent and slower blinks.	• High PERCLOS; • Slow blinks.

Researchers from a different study [80] chose to use a group of subjects where none of the elements had driving experience but all of them had skills whiles operating a computed, in particular video games. Before the experiment, each subject was trained until they could drive for 15 minutes without errors. After

[4]Percentage of Eyelid Closure
[6]*Alpha*-waves
[6]*Theta*-waves

training, a Neuroscan system [53] (with EEG, ECG and EOG) was used to monitor the simulated highway driving experience.

After the test, the mean fatigue levels, reaction times, and error percentage of the group had increased. The PSD[7] of the EEG signal showed a minor peak on the 5 Hz frequencies and a larger one on 10 Hz, both in multiple cerebral regions. The mean ApEn[8] of the ECG signal also increased, as well as the PSD of the HRV [9] on the lower frequencies (0.04 to 0.15 Hz), with a decrease on the higher ones (0.15 to 0.4 Hz). These changes on the ECG signal are known to be directly related to the sympathetic and parasympathetic nervous systems, which are then related to the individual's arousal and drowsiness levels.

Although somewhat off topic, distractions while driving can be even more dangerous if the driver is drowsy or intoxicated. In order to analyse how the drivers deal with different distractions while operation a vehicle, researchers from Toronto [31] used an NADS MiniSim equipped with a tablet where the distractions where played and a eye tracking device. The results of this study showed that the drivers commonly reduce the vehicle's speed when voluntarily distracted, but not when the distractions were involuntary.

To support its results, the researchers classified an involuntary distraction as an activity started by third parties that deviates the driver's attention from the driving process. A voluntary distraction has the same concept but is the drivers that initiates it. This study also mentioned a classification process that divides that subject's attention selection into 2 dimensions, each one divided into 2 possible origin (table 2.2).

Table 2.2: Voluntary and involuntary distractions within the attention selection modes [76]

	Automatic	*Controlled*
Goal Driven	Habit	Deliberation (Voluntary Distraction)
Stimulus Driven	Reflex (Involuntary Distraction)	Exploration

As an attempt to decrease the number of road accidents related to distractions, another groups of researchers from Valencia [18] build an experiment that allowed

[7] Power Spectral Density
[8] Approximate Entropy
[9] Heart Rate Variability

to test the efficiency of human-machine interfaces without a visual feedback. One of the devices tested was a haptic pedal, being a haptic interaction a tactical experience related to the perception of force and vibration.

This pedal was tested as an emergency alarm to possible crashes an as a drowsiness alarm, here combined with a visual and auditory stimuli. The first test focused on the driver's reaction time to the alarm and the speed control after it, showed that, on average and compared to a visual stimuli, drivers react 300 milliseconds faster and that the safety distance was increased to 6.7 meters. The second test, using the subject's heart rate, respiration rhythm, and self-assessment, consisted on three hours of night driving while in sleep deprivation. The results showed an increase of the attention level after the first alarm, with a progressive increase on vehicle control as the alarms were repeated.

2.2 Sensors and Signals

All the studies mentioned before required some kind of measures related to the road/environment, vehicle or driver, most of the time combining, at least, two of them. These measures could be vehicle-based (driver's performance), behavioural, physiological or even subjective [64], being this last category highly susceptible to driver's though and usually difficult to verify with other measured data [58, 65].

2.2.1 Subjective Measures

Although subjective assessments are not reliable enough to provide a complete evaluation of a subject, the usage of a well developed test with very specific and closed-ended questions can, sometimes, be the definitive factor in his general state classification or can even be used to validate acquired data using another approaches [28]. Typically these measurements are made before, during and after the tests and are useful to give researchers a perspective of how and what was the subject's experience when answering the report.

To evaluate the sleepiness level of an individual, one of the most used scales is the *Stanford Sleepiness Scale* (SSS), described in table 2.3. This scale, which is used to quantify sleepiness into 7 progressive steps, has been shown to have a high correlation with performance in monotonous tasks and to be sensitive to sleep deprivation [30].

Table 2.3: Stanford Sleepiness Scale

Rating	*Degree of Sleepiness*
1	Feeling active, vital, alert, or wide awake
2	Functioning at high levels, but not at peak; able to concentrate
3	Awake, but relaxed; responsive but not fully alert
4	Somewhat foggy, let down
5	Foggy; losing interest in remaining awake; slowed down
6	Sleepy, woozy, fighting sleep; prefer to lie down
7	No longer fighting sleep, sleep onset soon; having dream-like thoughts
X	Asleep

Another widely used scale for self-rating sleeping evaluation is the *Karolinska Sleepiness Scale* (KSS), describe in table 2.4.

Table 2.4: Karolinska Sleepiness Scale

Rating	*Degree of Sleepiness*
1	Extremely alert
2	Very alert
3	Alert
4	Rather alert
5	Neither alert nor sleepy
6	Some signs of sleepiness
7	Sleepy, but no effort to keep awake
8	Sleepy, some effort to keep awake
9	Very sleepy, great effort to keep awake, fighting sleep

Although this scale is typically used to support more trusted measures (objective and subjective), some studies have shown that KSS score is directly affected by sleep deprivation [39, 77]. Another study have also evidenced a connection between the obtains KSS score and the standard deviation from the optimal path and eye blinking frequency [36].

The KSS has also validated with EEG signal's features and with behavioural indicators of sleepiness (e.g. reaction time) [39].

2.2.2 Behavioural Measures

As the individual become sleepy, their facial expressions and body movements also change. Such movements includes the subject's head's position, how open are his eyes, how much are they blinking or his yawn frequency. Through the use

of video cameras and multiple algorithms for video segmentation, face detection and feature extraction, behavioural measures based on motion traction have increasingly been used to detect levels of sleepiness.

Camera-based systems are typically very sensitive to the environment's conditions upon the footage capture, such as lighting, occlusion and reflections. Some systems use infrared illumination to try solving this issue. Other systems use 3D cameras to add depth information to the image analysis.

Even with all the right tools, measures based on an individual's behaviour may be misleading since his expressions can change by reacting to situations unrelated to fatigue.

Figure 2.2: PERCLOS: Percentage of eye closure

Although not reliable enough to be used alone [73], the eyes blink frequency, movement and PERCLOS[10], illustrated on figures 2.2, have always been used in research studies and can still contribute as a complementary input on fatigue detection [27, 59, 79].

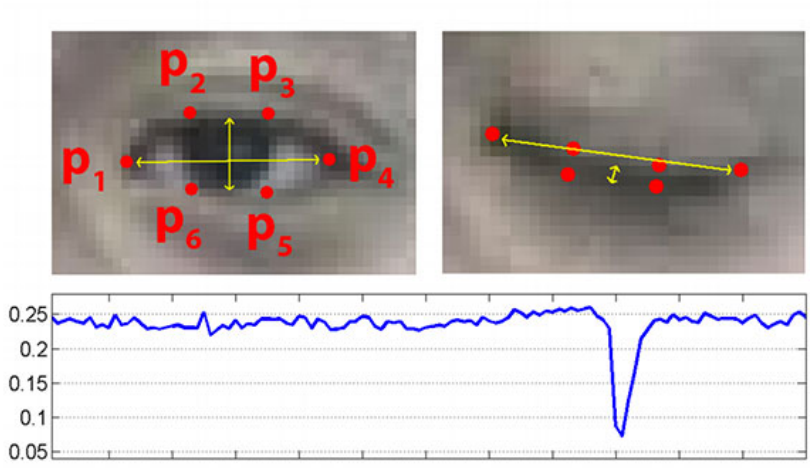


Figure 2.3: Eye blinking detection using the distance between eyelids

As the driver becomes more and more drowsy, his eyes blink frequency also increases and the speed of the movement decreases, which directly increases the

[10] Percentage of Eye Closure

PERCLOS value, acquired by measuring the distance between the upper and lower eyelid, illustrated on figure 2.3. However, fatigue detection is too complex to be detected only with this variable [73].

2.2.3 Driver's Performance Measures

Throughout the years, driver's performance measures, or vehicle-based measures, have been labelled as poor discriminating features for driver drowsiness for not being specific enough, since a driver under the influence of alcohol or others drugs would have a similar result.

Nevertheless, this approach to measure driver drowsiness have been boardly used [48], being one of the most common ones the steering wheel movement (SWM). When drowsy, the driver involuntarily reduces the number of micro-corrections (between 0.5 and 5 degrees) on the steering wheel. This measure can be hard to used in some situations due to being too dependent on the road geometry.

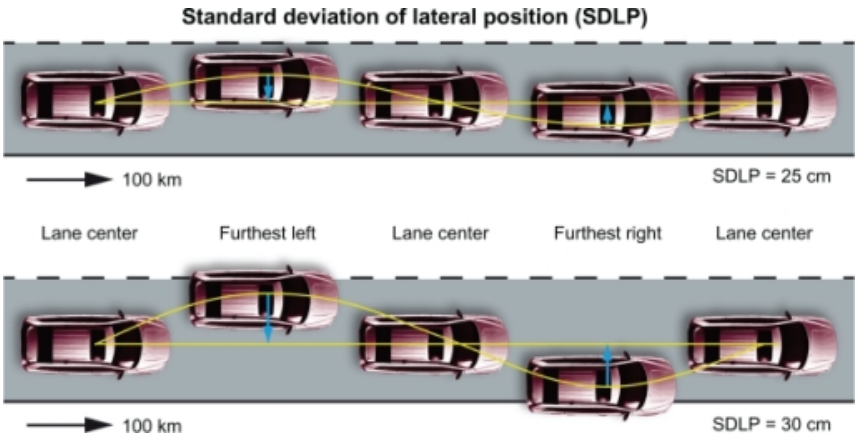


Figure 2.4: Ilustration of the measurement of SDLP

As the SWM reduces, it can be predicted that the vehicle's lane position will oscillate with increased amplitudes. The measurement of the standard deviation of this position (SDLP) is another measure widely used to evaluate the level of driver drowsiness. Although it has been related to KSS ratings, this measure also have significant dependency to the subject and road conditions, similarly to the previous measure.

Some vision based devices, such as the ones produced by *MobilEye* [52], can also be used to measure the driver's performance. This devices from *MobilEye* are

used on consumer level as ADAS[11] through system alerts of potentially dangerous situations but can also be integrated with other systems through a CAN[12] bus.

2.2.4 Physiological Measures

The last two types of measures only become detectable after the driver starts to sleep, which can be too late to prevent an accident. On the other hand, physiological measures, being signals directly acquired from the driver's body, appear to change in the early stages of drowsiness, making them more suitable to detect and alert a drowsy driver with enough time to prevent a possible accident.

There are multiple physiological signals that can be used to enrich an experience's data acquisition and give it more detail about a subject's state. Some of the most common are the electrocardiography (ECG), the electroencephalography (EEG), the electromyography (EMG) and the electrooculography (EOG) [10, 41, 42]. The photoplethysmography (PPG) and the electrodermal activity (EDA) are also measuring techniques that can be used for this purpose [19, 44, 45].

Although the usage of physiological signals directly contributes to a more complete acquisition system, they also increases its complexity level due to heavier signal preprocessing with multiple filtering phases. This need for better signal filtering comes from the fact that this type of signals have very low amplitudes with means that the Signal-to-Noise Ratio (SNR) is very low (e.g. the ECG signal voltage range is between 1 and 5 mV). After filtering, the signals are used in a feature extraction process which the result can then be used to evaluate the driver's performance [55, 64].

Being one of the most extracted features of the *ECG* signal (fig. 2.5), the heart rate can be described as the inverse of the time interval between heartbeats (RR intervals). The heart rate variability on the other hand (fig. 2.6), describes the variations in the RR interval times.

The power spectral density of an ECG signal, extracted using frequency-domain signal processing algorithms, describes the signal's power distribution along its frequency range. Some researchers consider the low frequency (LF - between 0.04 and 0.15 Hz) and high frequency (HF - between 0.15 Hz and 0.4 Hz) bands reliable

[11] Advanced Driver Assistance System
[12] Controller Area Network

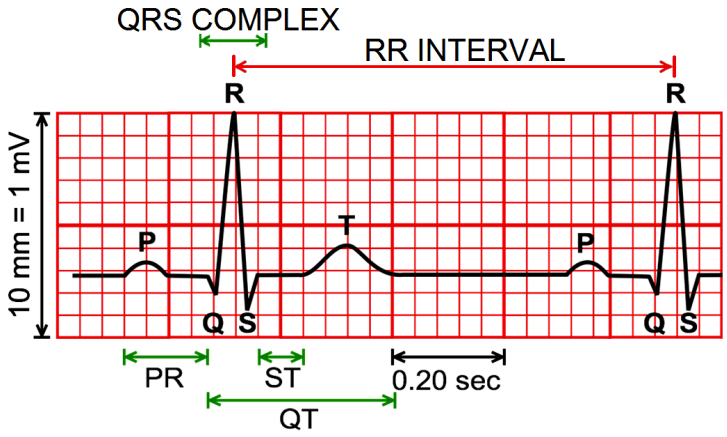


Figure 2.5: Illustration of a waveform of an heartbeat

measurements of the sympathetic and parasympathetic systems, respectively [46, 55].

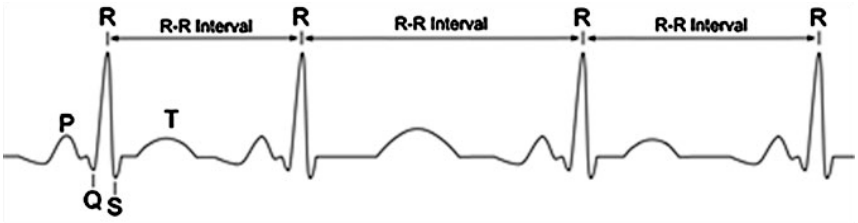


Figure 2.6: Example of the variation of an heart rate signal

The sympathetic nervous system's activity is known to respond to stress situations such as fight or flight and is also related to an individual's alert state. On the other hand, the parasympathetic nervous system, that has shown to be responsible for decreasing the heart rate and blood pressure, increases its activity while the individual is in a relaxed state of mind [75]. This two systems can contribute to the detection of the transition from being "awake" to be "sleeping" by using the HF and LF bands powers that, together, can be described as the LF/HF ratio.

By processing the ECG signal in the time domain, it can be extracted statistical features such as:

- SDNN - The standard deviation of NN intervals (regular RR intervals);
- SDSD - The standard deviation of the difference between adjacent NN intervals;
- The number of continuous intervals that differ by more than 50 milliseconds;

- PNN50 - The ratio between the last feature and the total number of NN intervals acquired.

Being the most used physiological signal to measure sleep levels, even being at the core of a sleep study, the EEG signal can be divided in multiple frequency bands, as shown on table 2.5.

Table 2.5: Common signals (brainwaves) extracted from EEG signal

Wave	*Frequency (ν - in Hz)*
Delta (δ)	$\nu \leqslant 4$
Theta (θ)	$4 < \nu \leqslant 8$
Alpha (α)	$8 < \nu \leqslant 14$
Beta (β)	$14 < \nu$

As in the ECG signal, the EEG can be processed in the frequency domain in order to calculate the power spectral density of each wave. Some studies have shown that the β waves activity, related to an individual awareness state, decreased after being in sleep deprivation or being monotonously driving for a long time. The θ and α waves also changed after the experiment [10].

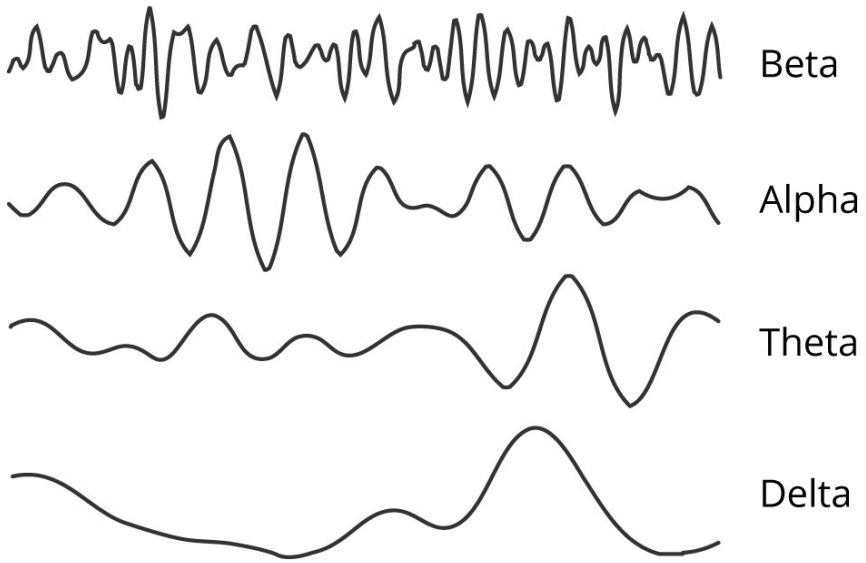


Figure 2.7: Illustration of 4 waves in which the EEG can be divided

2.3 Driving Games

"There's a massive difference between a game and a simulation, and yet there's a fine line that separates them" - James Dover from *Lets Race* [61] at TechRadar [54]

The main difference between a simulator and a game is that the aim of the simulator is the to get the best realism possible regardless of whether the experience is fun or not. It tries to get this features by modelling real world conditions such as physics and environment changes with much more precision than the entertainment games, which puts emphasis on providing the user with fun moments even if it means making the experience less real.

With continuous increase of the processing power of CPUs and GPUs (Central Processing Unit and Graphic Processing Unit), the difference between simulators and games are less and less visible, being more prominent in AAA games (top sale high quality games), as some of the mentioned on the next paragraphs.

In 2005, the *Image Space Incorporated* [35] developed *rFactor*. This game stood out for its approach to the modelling of the physics of tires and aerodynamics. These features were complemented by the support of mods, which allowed the customization of the game and its consequently adaptation to be used in professional simulations.

A seconds title of the franchise was launched in 2009 with an even better physics engine, improved graphics and a new weather system. Due to its realism, this game is still used to train professional car drivers and to test real component prototypes before them to be installed in cars.

From the italian studios *Kunos Simulazioni* [71], the *Assetto Corsa* was launched in 2014. This driving simulation game impressed most of the public by putting together high quality graphical environments and realistic physics simulation. On the other hand, the original artificial intelligence system, later updated to solve these problems, was very simple, monotonous and non aggressive, which are required features on a racing game. Sometimes the opponents reacted as if the user were not on the track, touching the user's car. Due to the high driving dynamics, the user's car could become unstable and could make the gaming experience much harder.

Another game that amazed most of the driving games fan base was *Project Cars*, launched in 2015 by *Slightly Mad Studios* [74]. It was considered by many the best

(a) rFactor

(b) Simulator running Assetto Corsa

(c) Project Cars

(d) iRacing

Figure 2.8: Examples of some highlighted driving games

driving simulation game for gaming consoles for its good physics engine (not as good as the *rFactor*'s one), very good graphics. The artificial intelligence system can be too aggressive with very little consideration for the player's position on the track and are also very erratic, braking and steering sideways for no apparent reason. This studio is also the creator of the *Need For Speed: Shift* games, which had a very similar technology that changed the franchise in the following years.

A different approach was made in the making of the *iRacing* game, published by *iRacing.com Motorsport Simulations* [37] in 2008, focused on the detail of the tracks, which are laser-scanned models of their real-world correspondents. Using this technique, the studios claimed that the product was a competitive game that could emulate with maximum detail a car race, making the gaming experience closer to reality than expected. The decision to make the simulator this detailed led to the partnership of the studio with organizations such as NASCAR, McClaren and Williams F1 teams so that it could be used as a training tool.

Moving away from the closed-circuit race tracks, the *Richard Burns Rally* game was published in 2004 by *Warthog Games* [8] and stand out from the other simulators due to its realistic physics motor. It has been considered by many as one

of the most difficult and realistic race simulators and continues to challenge the recent rally simulators, including the famous *Colin McRae Rally* franchise, due to its customization capabilities using mods published by the community.

Not being a race simulator, the *City Car Driving* [21] was published in 2016 and tries to focus in the educational purpose of a driving simulator, being the closest to what this thesis aims to achieve. It tries to give to the user a good driving experience in large cities or in countryside while also having weather simulation.

(a) Richard Burns Rally

(b) Euro Truck Simulator 2

(c) City Car Driving

Figure 2.9: Examples of some highlighted driving games

Although the graphic aspect of the simulator is quite low compared to the previously mentioned games and to the possible in 2016, the developers of the game focused on exposing the user to a realistic traffic situations to challenge him and to put his driving capabilities to the test. It also has a good physics engine and a

well defined traffic-rules system.

Another game that focus on the city driving simulation is the *Euro Truck Simulator 2*, launched in 2013 by *SCS Software* [6]. It gives to the user the experience of driving a transportation truck. Besides needing to obey the traffic rules, the user should also take breaks in the right time to reduce the driver's simulated fatigue.

3

System Overview

This thesis aims to develop a driving simulator system that will run on a low-level configuration (mentioned on section 2.1). This simulator will be used to acquire data related to the driver's performance while driving on a highway environment along with traffic.

Together with this performance data, it will be acquired the driver's ECG signal (mentioned on section 2.2) using a third-party system called *CardioWheel* embedded on the steering wheel.

Another requirement of this thesis was to have all the sub-systems of the project as independent modules that could be hosted by different machines. The support to this feature was added by developing an network communication architecture that allows the flow of information between all the modules. This architecture has at its core a module called *OSCD* and is illustrated on figure 3.1.

Both the simulator and the *CardioWheel* system are modules of this communication architecture. All the data acquired by them are transmitted through the system and registered on a database hosted by a fourth module. This database (relational model illustrated on figure 3.2) is divided into two tables: one for the message subject and timestamp and another for all the arguments transmitted on that message.

Both the communication infrastructure and the 3D simulator will be described in detail in chapter 4 and 5, respectively.

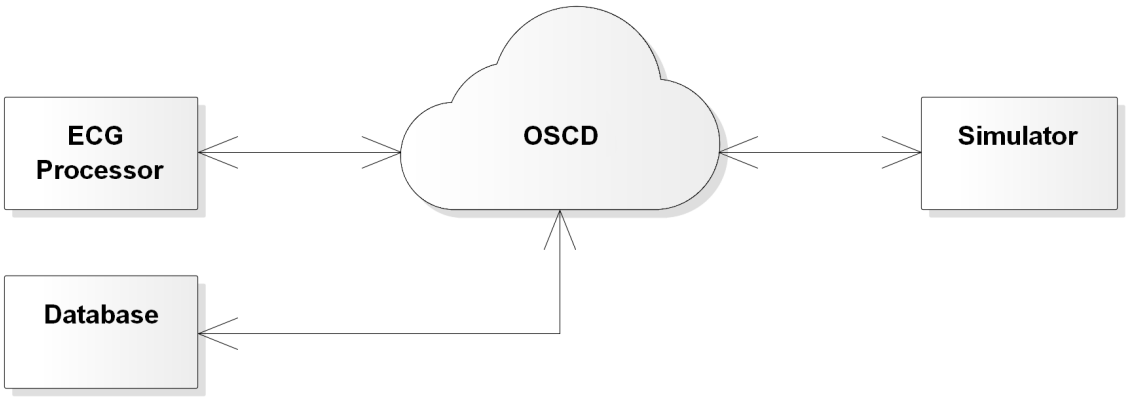


Figure 3.1: System Communication Infrastructure

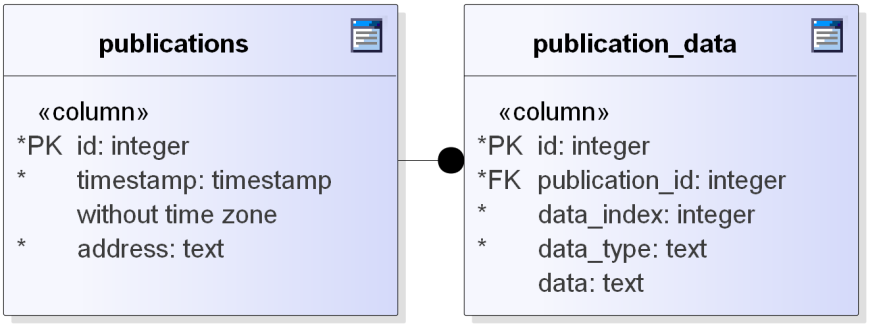


Figure 3.2: Database Relational Model

Two of the scenarios where the simulator is planned to be used is to acquire data while the driver is in sleep deprivation and while he is under the influence of alcohol. This two experiments will be used to classify driving while fatigued as an DUI[1] equivalent [17, 57].

Based on the concept of "Serious Games" [20], which use design principles of entertainment games for educational purpose or for training real life situations, it is intended to develop this simulator using the *Unity 3D* platform [7]. This game engine was chose because it is cross-platform compatible and has been highlighted in the educational field for allowing the development of advance solutions through a relatively easy environment. According to the *GameDesigning* blog [25], the *Unity 3D* platform is the 2nd most popular video game engine of 2017.

While the *Unity 3D* platform uses *C#* as its scripting language, the *CardioWheel* system is implemented in *Python*. The possibility of having multiple programming languages or even multiple versions of them makes it much more difficult

[1] Driving Under the Influence

to group all the technologies into the simulator's module itself. To solve this problem, each component would be encapsulated in distinct modules that would exchange data with the system. There are multiple approaches to achieve inter-process communication (IPC) but it is important to keep in mind the system's requirements:

- All clients must be able to exchange message with each other;
- A single message must be able to be received by multiple clients;
- The system's modules may or may not be running on the same machine (i.e. network communication);

With these requirements, some valid approaches would be *sockets* or *message queues* [33, 67].

3.1 Communication Paradigms

The usage of message queues allows the asynchronous message exchange between decoupled clients. If some client fails the whole application won't be take down and the pending jobs can be retried, which guarantees that the queued jobs will always be completed. Finally, message queues are highly scalable and can provide performance profiling in order to provide issues [38, 49].

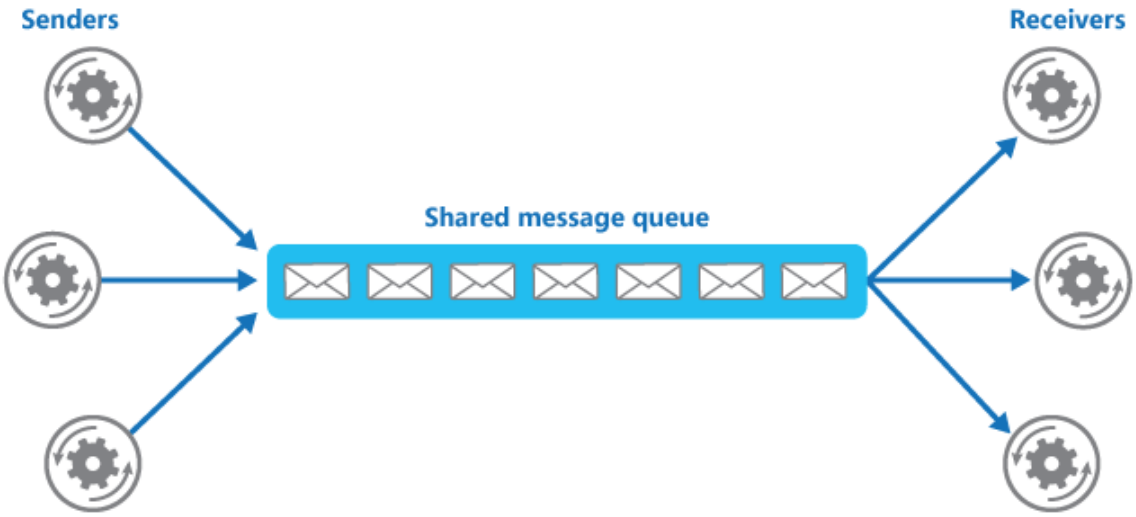


Figure 3.3: Illustration of messaging using message queues

This approach to messaging doesn't come without disadvantages, being one of them the fact that all clients of the queue have access to all queued messages and that each job must wait its turn before being dequeued.

Alternatively, sockets are efficient, can provide almost full decoupling, can use TCP or UDP and can integrate easily with event-driven programming. However, its implementation can be very complex and time consuming.

Another advantage of socket usage is that any TCP/UDP-based protocol can be used. This enables the usage of different protocols and techniques which allows the usage of many types of integration patterns [32].

3.2 Integration Patterns

Because the project consists of several modules that may or may not exchange data with each others and may or may not be hosted by different environments, those modules will have to communicate through a network using a architecture that can route the data to the correct destination.

There are some integrations patterns that can be used to distribute the information, such as the *Publisher-Subscriber Pattern* and the *Message-Broker Pattern*. The first pattern uses an infrastructure that allows the publication and subscription of messages in order to distribute the information only to the clients that subscribed to it.

The *Publish-Subscribe* pattern [51] can be implemented in multiple ways, such as:

- Lists - by using something that can identify the subjects (e.g. subject's description) stored and maintained in a list with all the registered subjects, a given subject can be redirected to all its subscribers;
- Broadcast - this approach uses the broadcast address of the network to distribute the message between the clients. This means that every machine connected to that network will receive the message, even if not part of the distribution system. Each client must analyse these messages to validate if it is subscribed to it.
  This method has the disadvantage of making excessive and uncontrolled use of the available bandwidth and therefore can be easily blocked by network management devices;

- Content - This implementation approach to the *Publish-Subscribe* pattern, being the most recent of the three mentioned, differs from the rest by forwarding messages not using the subject's description (topic-based) but rather through the analysis of the message's content (content-based). Subscribers using this method describe what kind of data they want to received instead of using its description. This makes the systems that use it more flexible.

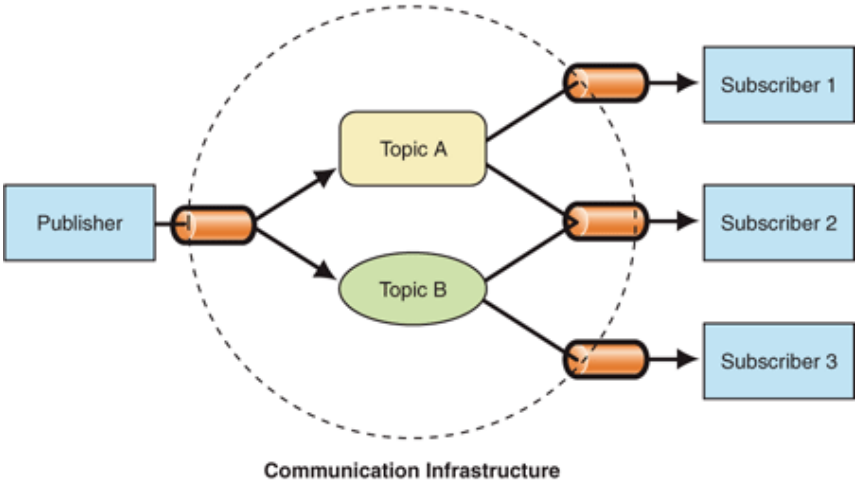


Figure 3.4: Example of a communication infrastructure using the *Publish-Subscribe* pattern

Implementations of this pattern may also use different methods to subscribe to subjects, either **static** or **dynamic**. In a system with static subscriptions these are defined by the development team and it is not possible to manage them afterwards. On the other hand, a system with dynamic subscriptions, although it can have initially-defined subscriptions, it is possible to remove them and add new ones in execution time (figure 3.5).

In addition to the features defined, implementations of this pattern should also defined how to discovery available subjects and if the system supports the use of *wildcards* subscriptions, which allow the subscription of multiple subjects using a generic query.

As shown in figure 3.4, if a communication infrastructure uses the *Publish-Subscribe* pattern with the *list*-based or the *content*-based approach, it uses some type of *Message Broker* [50]. This integration pattern uses a middleware that received, translates (if needed) and forwards messages to the specified destination.
In fact, both the patterns share some advantages and disadvantages:

- Advantages

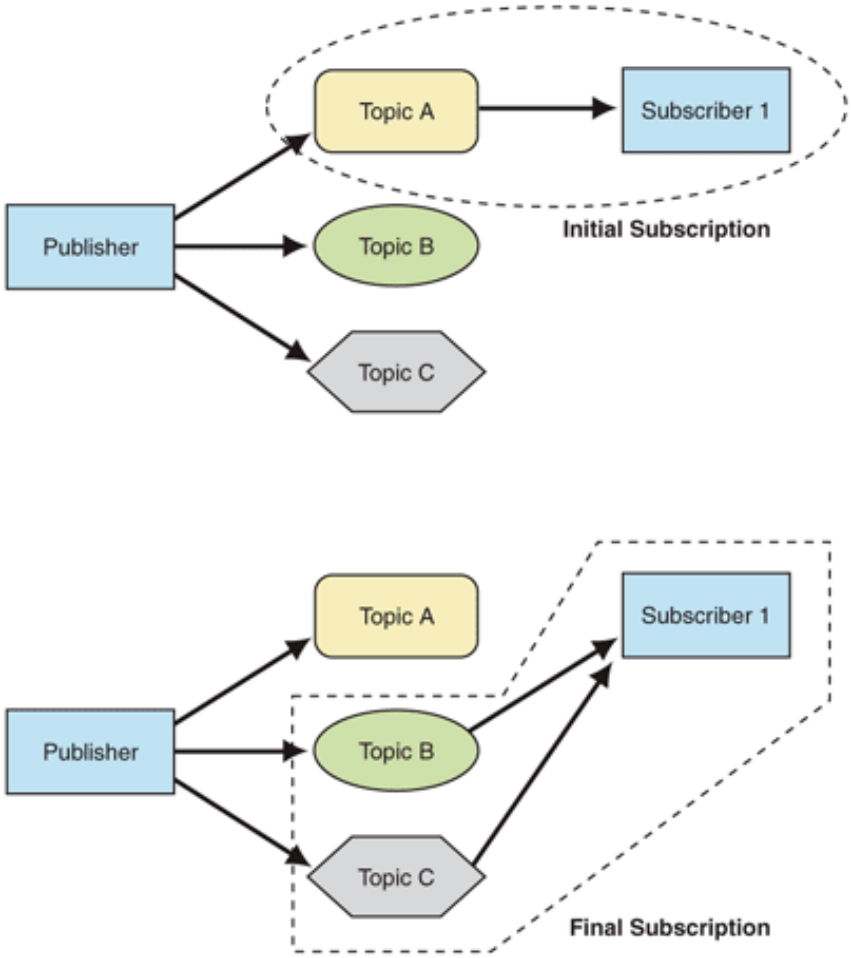


Figure 3.5: Example of the *Publish-Subscribe* with dynamic subscriptions

- Coupling reduction - a publisher does not have to know how many subscribers does it have or who they are. This also applies to the subscribers;
- Security improvements - Since the communication infrastructure is responsible for the coordination and routing of messages, only modules subscribed to a certain subject will receive it;
- Increased testing capacity - for being topic-based, the number messages requiring testing is reduced;

- Disadvantages
  - More complex - it must have an extra system to manage all the subjects;
  - Lower performance - subject management and query operations will increase the data transmission latency, which slows down the forwarding process.

3.3 Communication Protocol

Having the OSC[2] initially been developed to replace the MIDI[3] protocol for music data transmission between instruments and computers, it quickly began to be used in projects where a fast and simple data transmission were crucial [78].

Although it is a very straightforward protocol, the OSC is message-oriented and it has the advantages of being interoperable, precise, flexible, organized and, at the same time, easy to implement. There is no restriction on which protocol is used to transport the data, being TCP (*Transmission Control Protocol*) and UDP (*User Datagram Protocol*) the most common ones.

In OSC protocol, there are 2 types of data packets (*OSC Packet*): *OSC Messages* and *OSC Bundles*. The base structure of the *Messages* is:

- *OSC Address Pattern* - symbolic name that follows the URL (*Uniform Resource Locator*) concept. Its value must be at least one sequence of the "`/`" character followed by a name that should be hierarchical, expressive and potentially self-documenting.
  The address can also be a *wildcard* by using the special characters `?`, `*`, `![-]` and `{,}` for a partial address match. Using the figure 3.6 as context, the "`/SIMON/Simulator/*`" address would match with the "`/SIMON/Simulator/PathOffset`" and the "`/SIMON/Simulator/ReactionTime`" addresses;
- *OSC Type Tag String* - begins with a "`,`" character and is used to identity the data type of the arguments in the message. A usage example of this string would be "`,ibb`", which indicates that the current message has one 32-*bit* integer and two *blobs*. Some types are self-described which means that do not have a specified value;

[2] Open Sound Control
[3] Musical Instrument Digital Interface

- Arguments - the actual values of the data that was identified in the *OSC Type Tag String*.

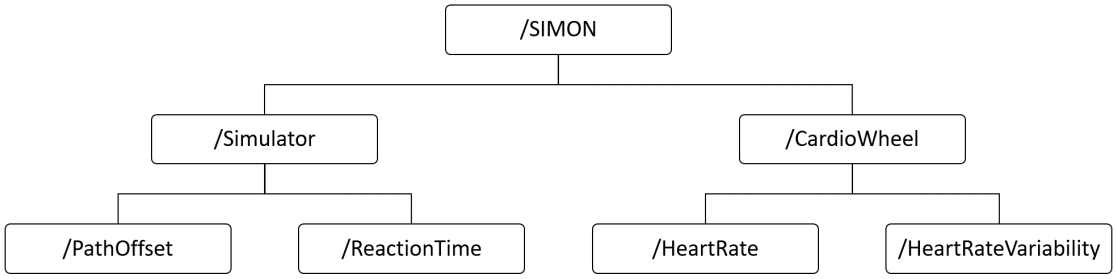


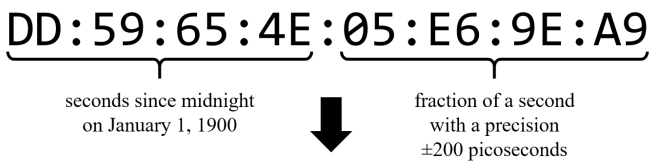
Figure 3.6: Example of an *OSC Address* hierarchy

The 1.0 specification of the protocol defines 14 data types: four required and ten optional. Table 3.1 lists these data types and the figure 3.7 gives more detail about the composition of a *OSC Time Tag*.

Table 3.1: OSC 1.0 defined data types

	Tag	*Data Type*
Required	`i`	32-*bit big-endian*[4] integer
	`f`	32-*bit big-endian* floating point number
	`s`	Sequence of non-null ASCII characters followed by a null
	`b`	(Blob - *Binary Large Object*) Begins with its size stored as a 32-*bit* integer, followed by that many 8-*bit* bytes of arbitrary binary data
Optional	`h`	64-*bit big-endian* integer
	`t`	64-*bit big-endian* fixed-point time tag (figure 3.7).
	`d`	64-*bit big-endian* floating point number
	`c`	ASCII character (32 *bits*)
	`r`	RGBA[5] Color (32-*bit*)
	`m`	MIDI message (32 *bits*) with [*Port ID*, *Status*, *Data1*, *Data2*]
	`T`	*True* boolean value. No argument data
	`F`	*False* boolean value. No argument data
	`N`	*Null* value. No argument data
	`I`	*Infinitum*. No argument data

[4]The most significant byte is stored in the smallest address
[5]RGBA - *Red, Green, Blue, Alpha*



September 5th, 2017 at 20:09:14,098999977111

Figure 3.7: Composition of a *OSC Time Tags*

NOTE: An *OSC* packet must always have its size multiple of 4 bytes (32 bits). To ensure this restriction, any data argument with a non valid size will be followed by a 0-3 byte padding (null characters).

4

Architecture

In the last chapter it was mentioned some of the reasons that led to the decision of implementing an messaging architecture using the *OSC* protocol and the *Publish/Subscribe* pattern. The distribution of a message to several modules and the possibility of executing these modules without dependency to others, were the main motivators to developing the *OSC Distributor* (OSCD) architecture, including a protocol to enable the management of clients in the system.

Since the proof of concept of this thesis will be focused on the monitoring of subjects while driving with some level of fatigue or under the influence of alcohol, it is necessary to have data extracted from the subject's ECG signal, described on section 2.2.4. To get this type of data it was used the *CardioWheel* device in communication with a modified version of its server.

4.1 OSC Distributor (OSCD)

The central module allows the registry of users, using a simplified approach, as well as the registry and subscription of subjects requested by those clients and has some extra features.

This server would need clients to communicate with and so, to test the implemented solution, a C# client was implemented in parallel with the server's development.

This *Publish-Subscribe* patterned system will support the usage of dynamic subscription so the clients can add and remove subjects and subscriptions. It does also support the usage of a dedicated message to discovery the available subjects (described on the next section) but will not subject wildcards, which means that there is no way to add multiple subjects with a more generic one.

4.1.1 Server-Client Protocol

The *OSCD* protocol establishes that the server must respond to the client's commands in most cases. This response can be whether positive (using the `/OSCD/Response` OSC address) or negative (i.e. an error - using the `/OSCD/Error` OSC address). All the errors that the server can respond with are complemented by a code based on the *HTTP*[1] *status code* [22]. The ones used are:

- *400 Bad Request* - used when the message received is in some way malformed;
- *401 Unauthorized* - used when the message received includes an `userID` parameter but it does not match to the one registered on the server. It can also be used if the user tried to execute a registration-protected command without having one;
- *404 Not Found* - used when the server receives a command with an unknown address;

All the commands sent by the client must have a `commandID` parameter that the server will then use as parameter of the response message. If the command is protected by registration, then the client must also include the `userID` parameter given by the server upon the client's registration.

***Register User* Command**

Typically, the client should first register it self on the system by sending a *Register User* command to the server. When the server receives this command, it creates a new client based on the TCP connection from where the message was received. Without it, the client is only allowed to execute the *Get Available Subjects* command (described later in section 4.1.1).

The server's response to this command includes the new `userID` of the client in the system and it should be saved and used in the next commands.

[1] Hypertext Transfer Protocol

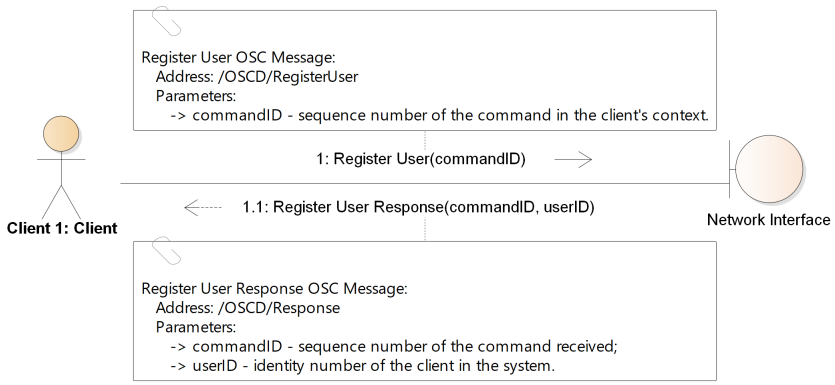


Figure 4.1: Communication model of a *Register User* request

Register Subject Command

After adding the `commandID` parameter, the client must also include its `userID` and one or more subjects that it wishes to register on the *OSCD* system.

The server responds with the same number of network ports that it received of subjects. Each port is defined with de UDP protocol and indicates where each message of each subject should be sent.

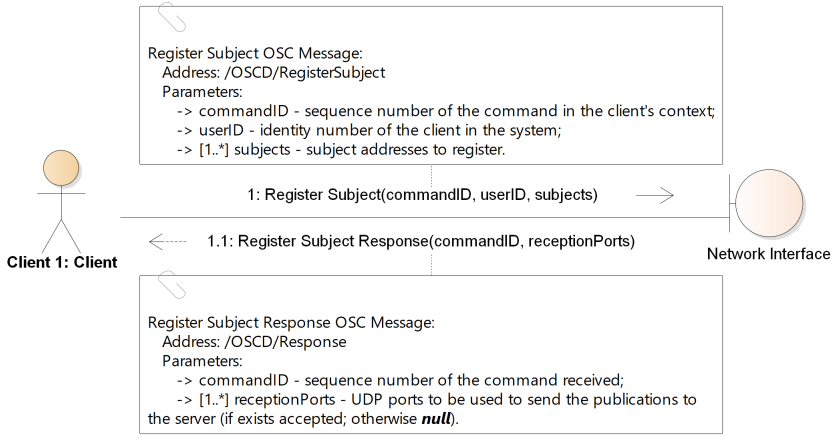


Figure 4.2: Communication model of a *Register Subject* request

Subscribe Subject Command

To subscribe a subject, the client, besides including the parameters `commandID` and `userID`, must add a triplet per subject that it want to subscribe. This triplet must have as first entry the subject to subscribe, the second one the UDP port to where the subject should be sent by the server and on the third entry, a notification flag. This flag indicates to the *OSCD* server if the client wants to enable the subject notification if it is not available.

The response received from the server includes as many boolean flags as there were subjects in the request. This flags indicates if each subjects was subscribed.

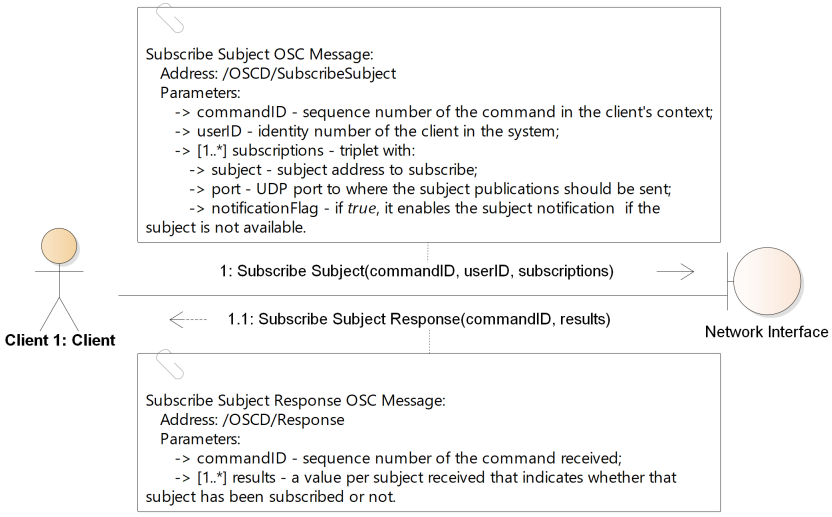


Figure 4.3: Communication model of a *Subscribe Subject* request

Get Available Subjects Command

Similarly to the *Register User* command, to send a *Get Available Subjects* request, the client only needs to include the `commandID` parameter. The server responds with a list of all subject addresses that are registered on the *OSCD* server.

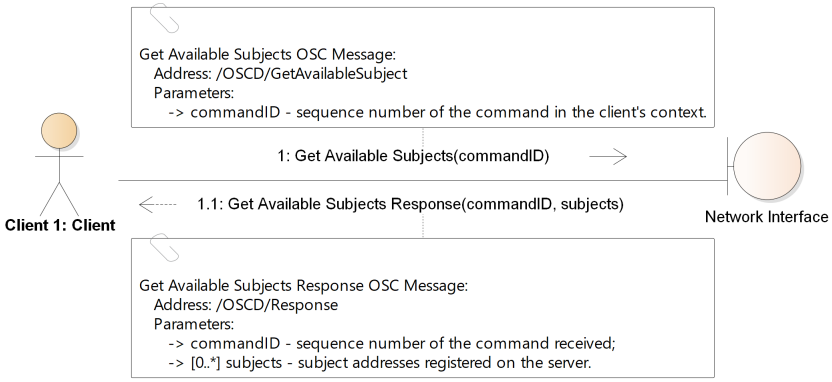


Figure 4.4: Communication model of a *Get Available Subjects* request

Unregister Subject, *Unsubscribe Subject*, *Enable Subject Notification* and *Disable Subject Notification* Commands

In order to remove a subject registered by it, the client must include the subject address with the "standard" `commandID` and `userID` parameters. This request

has no response since that can only be two outcomes: the client has registered the subject sent and that subject is removed or the client has never registered that subject and the server ignores the request. In both the possibilities, the client "no longer" has the subject registered.

The models of the other 3 commands are available on appendix B since the message semantics are the same. As expected, the OSC addresses used in this *OSCD* commands are:

- *Unregister Subject* - `/OSCD/UnregisterSubject`;
- *Unsubscribe Subject* - `/OSCD/UnsubscribeSubject`;
- *Enable Subject Notification* - `/OSCD/EnableSubjectNotification`;
- *Disable Subject Notification* - `/OSCD/DisableSubjectNotification`.

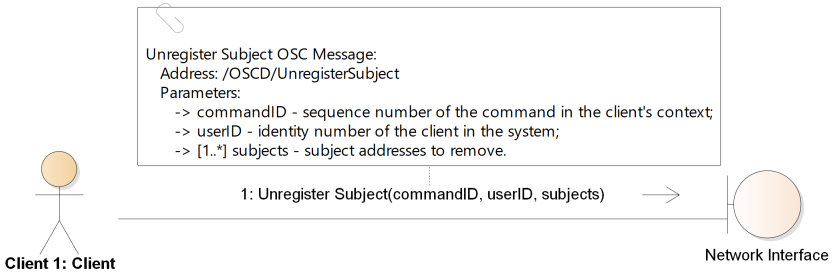


Figure 4.5: Communication model of a *Unregister Subject* request

Subject Publications

To publish data in a subject context, the client must include its `userID` first followed by all the data that it wishes to publish. This message should be sent to the *OSCD* server using the UDP port received from it when the subject was registered.

The server will not respond to the subject's publisher but will forward the message to all the clients that subscribed to it.

Subject Notification and *Subject Cancellation* Messages

Both the *Subject Notification* and *Subject Cancellation* messages are not requests that the client can make to the server. This commands are used by the server to notify

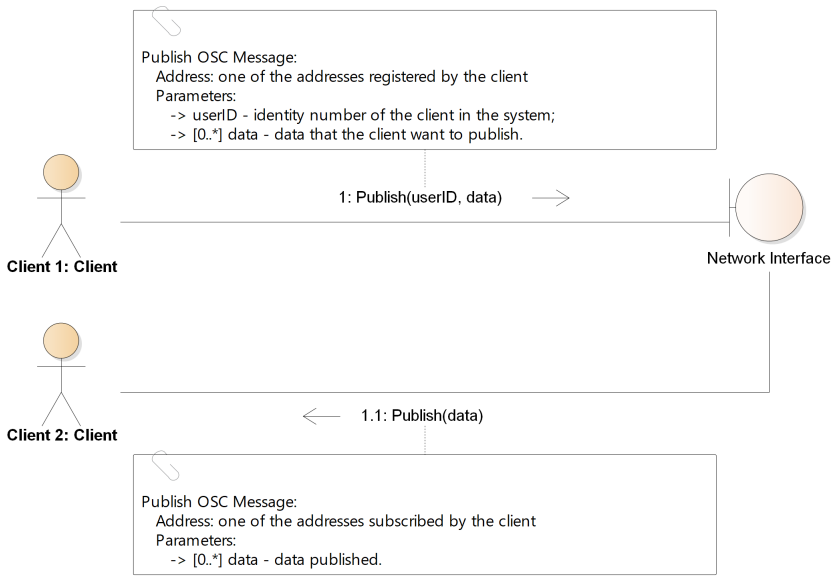


Figure 4.6: Communication model of a subject publication

the client when a subject that has the availability notifications enabled becomes active (i.e. when other user registers it) and when a subscribed subject becomes obsolete (i.e. when the user removes it or disconnects from the server).

The client should always be listening to this two commands in order to react accordingly.

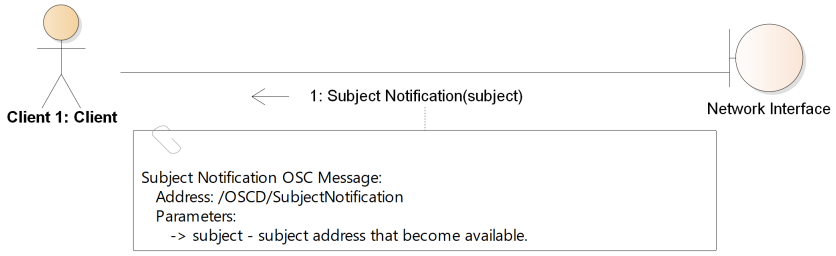


Figure 4.7: Communication model of a *Subject Notification* message

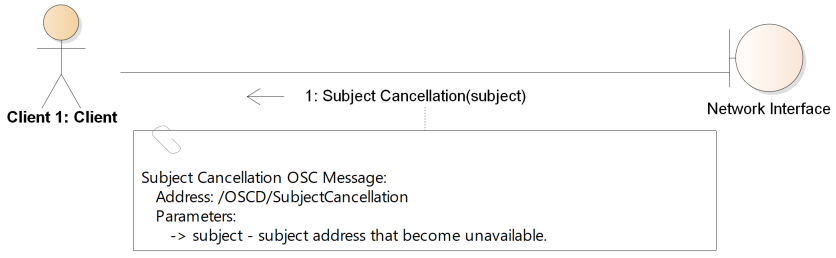


Figure 4.8: Communication model of a *Subject Cancellation* message

4.1.2 `Core` Package

The message distribution center started with the implementation of the user management system which includes the classes `OscSubject`, `OscUser` and `OscDistributor`. This package does not yet have the concept of network.

Figure 4.9: (Simplified) Structure of the OSC Distributor manager (appx. C)

`OscSubject` Class

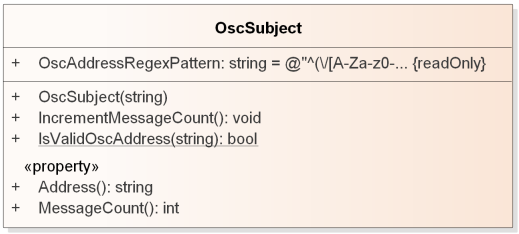


Figure 4.10: `OscSubject` class

This class represents a subject (OSC address) in the system. It keeps track of the number of messages sent with the respective address and it has a **static** method (`IsValidOscAddress`), together with a *regex* string (`OscAddressRegexPattern`), to validate them.

`OscUser` Class

Representing a user in the system, the `OscUser` class keeps track of the subjects registered and subscribed by it. The subjects registered (`subjects` attribute) are those with which the user wants to publish information and are stored in a `Dictionary` instance where the subject address is the *key* and the *value* is an `OscSubject` instance.

Similarly, the subscribed subjects (`subscriptions` attribute) are the ones that the user wants to receive publications. These subjects are also store in a `Dictionary` instance where the *key* is the subject address and the *value* is the network port where the user's client is expecting to receive publications.

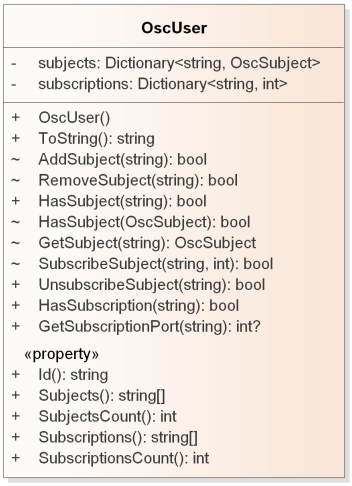


Figure 4.11: `OscUser` class

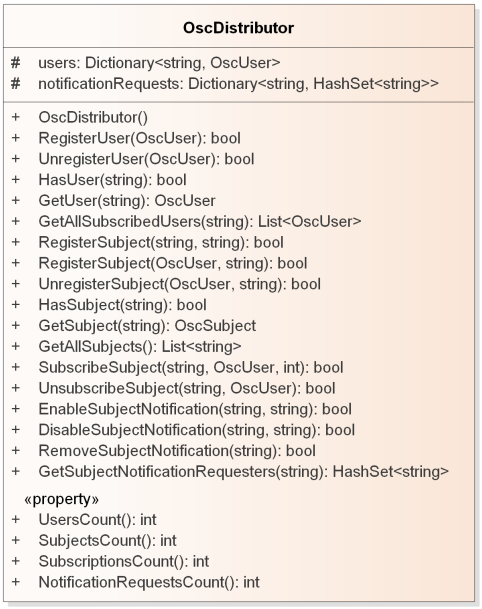


Figure 4.12: `OscDistributor` class

`OscDistributor` Class

Being one of the central piece of the architecture, the `OscDistributor` class allows the management of users, subjects, subscriptions and notification requests.

It stores the users in a `Dictionary` instance where the *key* is the user's ID and the *value* is an `OscUser` instance. The notification requests are also stored in a `Dictionary` instance where the *key* is the subject of the notification and the *value* is an `HashSet` instance with user IDs.

4.1.3 `Network` Package

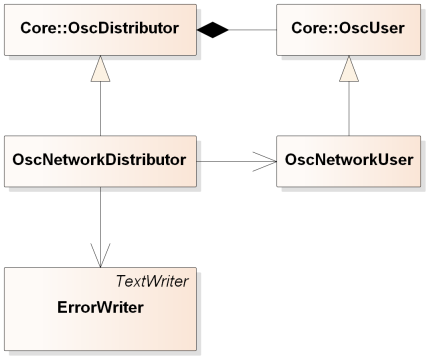


Figure 4.13: (Simplified) Structure of the OSC Network Distributor manager (appx. C)

`OscNetworkUser` Class

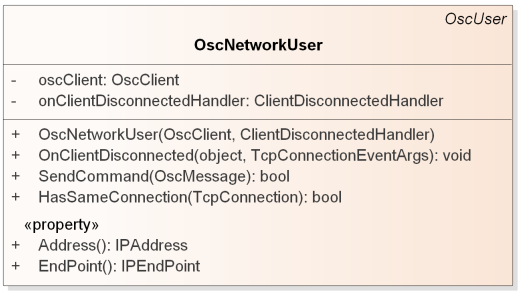


Figure 4.14: `OscNetworkUser` class

Being a specialization of the `OscUser` class, the `OscNetworkUser` class adds the support of an endpoint to an *OSCD* client and so the capability to send messages to it. It has also a delegate method (`ClientDisconnectedHandler` delegate) that is used when a disconnection event is listened. This allows the execution of clean-up instructions when a client is lost.

`OscNetworkDistributor` Class

The `OscNetworkDistributor` class is a specialization of the `OscDistributor` class that manages the network clients and that handles the *OSCD Protocol* communication and the subjects publications propagation.

When is received *OSCD Protocol* commands, the `OscNetworkDistributor` class almost always must to execute some instructions that will generate more commands that need to be sent to the remaining users:

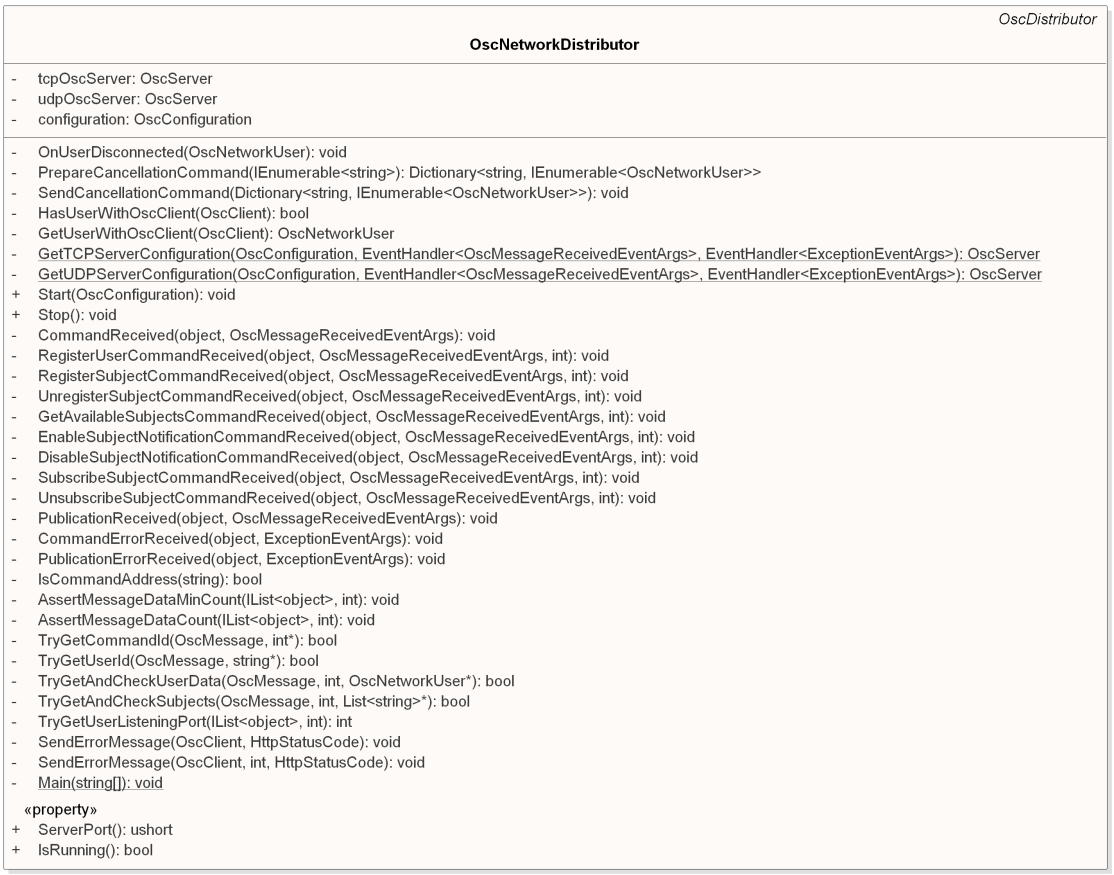


Figure 4.15: `OscNetworkDistributor` class

- *Register Subject* - when a subject is registered by a user, the distributor checks if any of the other users has already registered the received subject. If it has, then it sends to the user a *Response* command with a `false` boolean value. If not, it must check if any of the other users has requested an availability notification for the subject received. The notifications are them sent to all the users that requested it and the user that registered the subject receives a *Response* command with a `true` boolean value;

- *Unregister Subject* - if a user wants to delete a subject registered by him, the distributor must check if that subject has any subscribers. If it has, they must receive a *Subject Cancellation* command for the deleted subject;

- *Subscribe Subject* - the request to subscribe a subject can have a positive outcome or a negative one. If the subject has already been registered, then the distributor responds to client with a *Response* command with a *True* boolean. Otherwise, the distributor checks if the user wants to be notified when the subject become available and acts accordingly;

- *Enable Subject Notification* - when it receives a request to activate availability notifications, the distributor checks if the subject received is registered in the system. If it is, then it sends a *Subject Notification* command with the subject available. Otherwise, it adds the user the sent the command as a receiver of a future notification.

4.2 OSCD Clients

Since the *OSCD* server implements a message protocol, a client module is needed as an abstraction to the future consumers of the system. As both the C# and the Python clients have the same base structure and logic, the implementation description will be described in a general manner that, in the subsections 4.2.1 and 4.2.2 is language-specified.

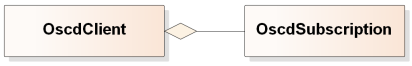


Figure 4.16: Generic (simplified) structure of an *OSCD* client

In order to reduce unnecessary network traffic, the *OSCD* each client keeps state of its pending commands[2], active subjects, active subscriptions and active notifications. This information is later used to filter actions executed by the user.

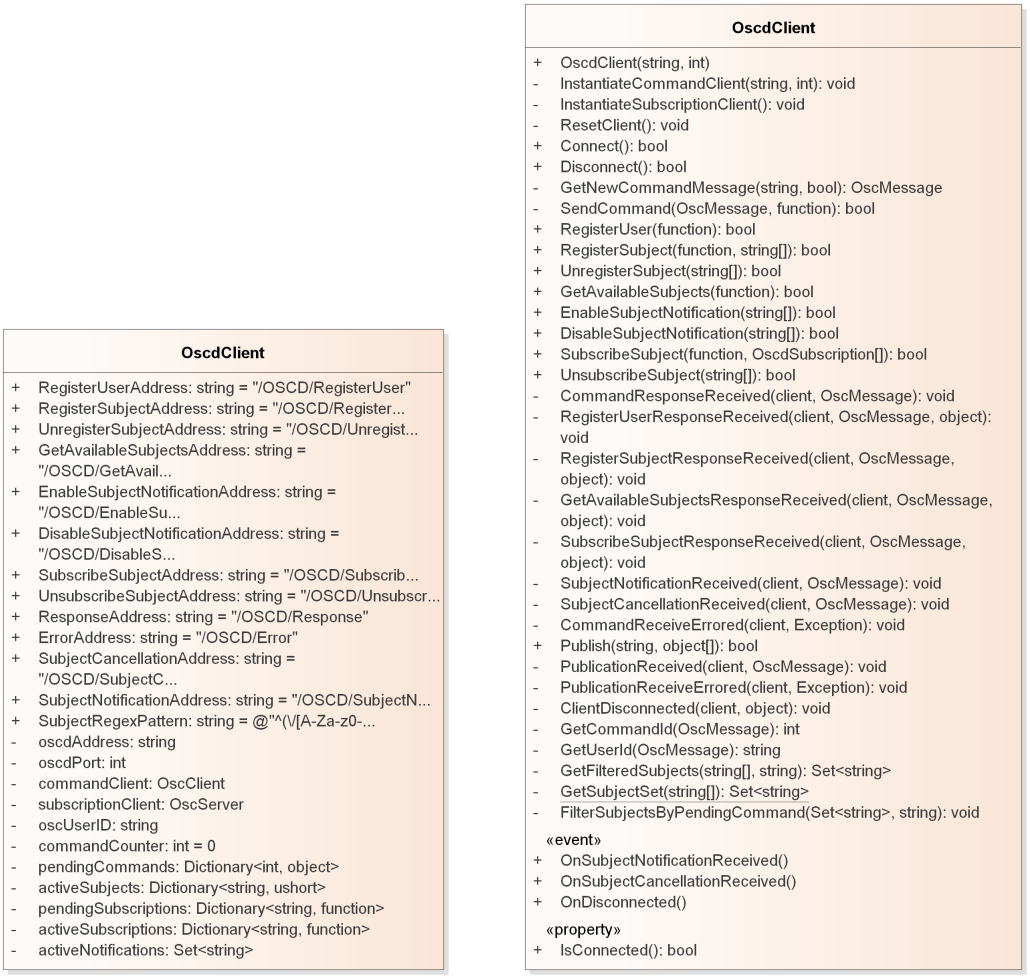
Whenever the user wants to send a command to which the server will respond, it needs to use as a parameter of the command's method, a function that receives and can handle the response message.

Although most of the commands only need the target subject(s), this does not apply to the *Subscribe Subject* command. Instead, this command receives instances of the `OscdSubscription` class that stores the subject, the function that will be used to process all the messages with that subject and a boolean flag that if is *True*, is used to activate the availability notification for that same subject.

Before sending the user's request to the *OSCD* server, the client:

1. Builds a set with the subjects received by the user. This removes all the duplicates and, if there was any, a exception is thrown;

[2]A pending command is a command that was sent to the *OSCD* server but didn't received a response yet.



(a) Attributes (b) Operations

Figure 4.17: `OscdClient` class

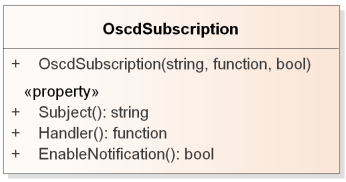


Figure 4.18: `OscdSubscription` class

2. Checks if the subjects are valid *OSC* addresses. If any of them isn't, a exception is thrown;

3. Removes the active subjects, active subscriptions and active notifications from the set obtained on point 1;

4. Finally, it discards all the subjects that are included in pending commands

of the same type.

To use the *OSCD* client, the user has also to define three functions that will be used to handle some other events:

- `OnSubjectNotificationReceived` event - is invoked when an availability notification is received from the server;
- `OnSubjectCancellationReceived` event - whenever a subscribed subject is cancelled, the server will send a cancellation notification so the client can handle it in the best way;
- `OnDisconnected` event - This event is invoked when the connection to the server is lost, regardless of the reason behind it.

4.2.1 *C#* Client

As predictable, the client implementation is not exactly the same as the one described at the beginning of this section. The implementation of the OSC network protocol was handle by a library with the name "*Bespoke OSC*".

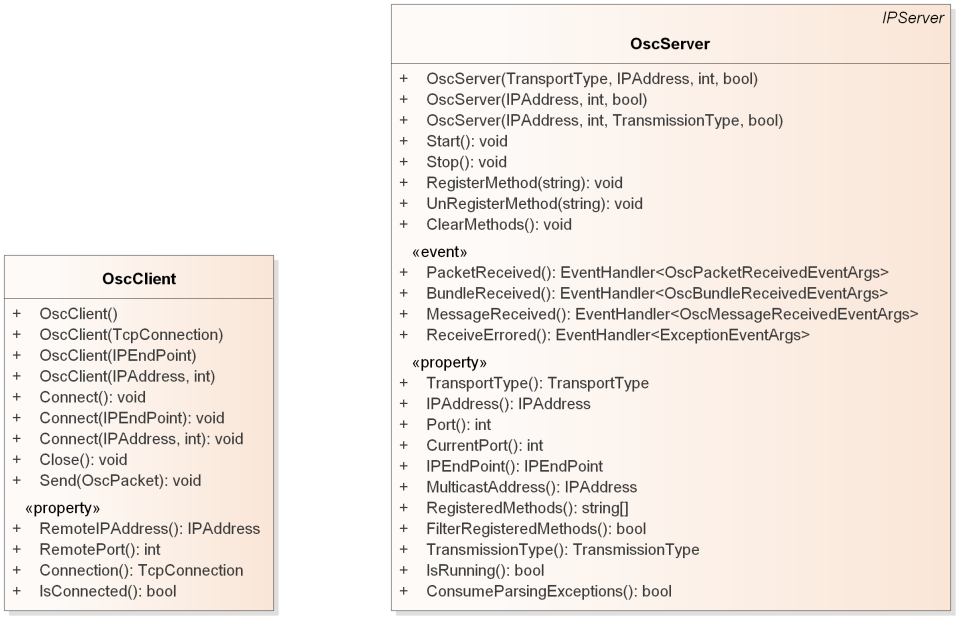
4.2.1.1 Modifications to the Protocol Library

Although this library had already an implementation of a client (`OscClient`) and server (`OscServer`), the first one only allowed to send information (`Send` method on Figure 4.19(a)) and the second to receive it (event handlers on Figure 4.19(b)). Since it was needed an object that could allow both, the class `OscBiDirectionalClient` was created.

The new client was implemented as a specification of the `OscClient` class, as most of the functionalities had already been implemented, and the some of the server's properties were also copied.

4.2.1.2 Event Handlers and Arguments

In C# it is possible to use *events* as a way for the class to notify a client that something happened. With this concept, the `OscdClient` class supports `EventHandlers` for the `OnSubjectNotificationReceived`, `OnSubjectCancellationReceived` and `OnDisconnected` events. The first two use the `SubjectAlertReceivedEventArgs` class as value and the last one uses the system's `EventArgs` class.



(a) `OscClient` class. (b) `OscServer` class.

Figure 4.19: Public attributes and methods of the *Bespoke OSC*'s client and server

Figure 4.20: Public attributes and methods of the `OscBiDirectionalClient` class

4.2.1.3 Auxiliary Classes

As a strongly-typed programming language, the implementation of the client in C# had to have auxiliary classes in order do keep the information flow organized. As so, the `CommandHandlerData` class (Figure 4.21) was created to store the information needed between sending and receiving commands and is used as values of the `pendingCommands Dictionary`. The only propose of this class

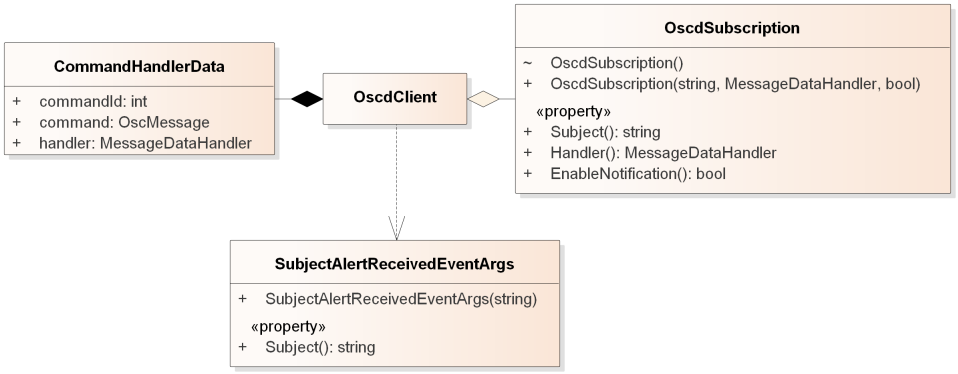


Figure 4.21: (Simplified) Structure of the *OSCD* client in C# (appx. D)

is to hold the *OSCD* command sent, its identification number and the callback function that will handle the response.

To allow the use a function as a callback, the `MessageDataHandler` delegate was created. This delegate defines a function that receives an array of `object` instances and returns `void`.

4.2.2 *Python* Client

Since Python features a dynamic type system, none of the client's attributes or functions have a predefined type. This means that there is no need to create auxiliary classes to hold information or event handlers and its attributes. This also means that the information flow is much more ambiguous and, for this reason, it was decided that the `OscdSubscription` class would be also implemented in this version of the *OSCD* client, since that is the only one that is received from outside the package.

4.2.2.1 Type Hinting

In order to maintain the information flows as strict as possible, *type hints* commentaries were used. This commentaries can have multiple formats and since the client was developed on *JetBrains*'s *PyCharm*, its *docstring* format was adopted.

As exemplified on Listing 4.1, the client's function `subscribe_subject` has 3 arguments: `self` (reference to the current instance of the class), `handler` (callback function that will handler the server's response) and `subscriptions` (list of subjects to subscribe).

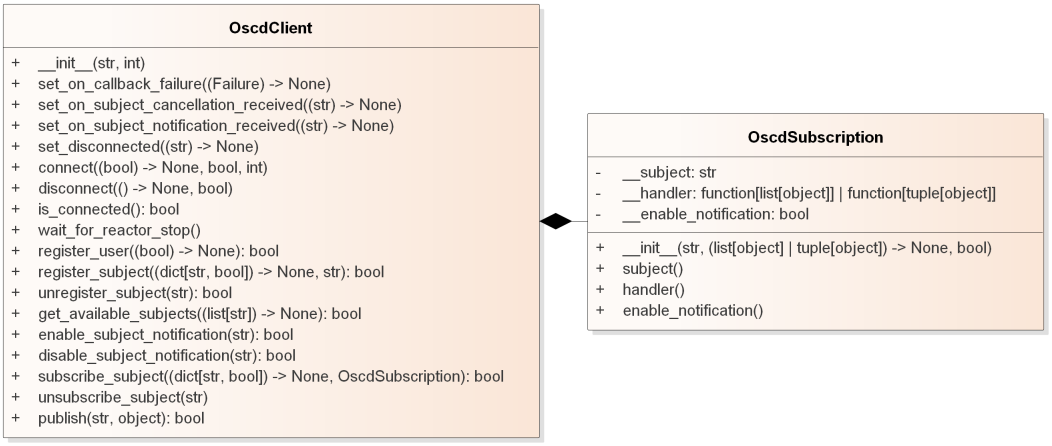


Figure 4.22: (Simplified) Structure of the *OSCD* client in Python

```
1   def subscribe_subject(self, handler, *subscriptions):
2       """
3       :type handler: (dict[str, bool]) -> None
4       :type subscriptions: OscdSubscription
5       :rtype: bool
6       """
```

Listing 4.1: *PyCharm docstring* type hinting example

Without the type hinting commentaries, the function signature would be too ambiguous but by defining that the `handler` argument is a function that should receive a *dictionary* and return nothing, it has now some information that can be later used by the *IDE* or by manual consultation. This information can be pushed even further by defining that the *dictionary* received by that `handler` function has *strings* as keys and *booleans* as values.

Although the type hinting of the function defines that the `subscriptions` argument is an `OscdSubscription` instance, the character "*" before the argument's declaration means that is actually a tuple of instances of that class. It also defines that the function returns a boolean.

4.2.2.2 *Twisted* - Event-driven Networking Engine

The *Twisted* framework has at its core an module called `reactor`. This module handles all the events that occur within the system and, in a basic manner, is a "infinite" loop. This loop starts when the `reactor`'s function `run` is called and nothing defined after the call will be executed until the module is stop. This framework will be defined in more detail on section 4.3.

This is a very restrictive approach since it requires the redesign of the software

architecture where the framework will be used. In order to prevent this, the `OscdClient` class executes the `run` function in a separate thread and uses the `connect` and `disconnect` functions to start and stop this thread. One limitation of this approach is that the thread and the `reactor` module can only be executed once per process.

4.2.2.3 Event Handlers - Callback Functions

In Python, to use a given argument as callback it only needs to be `callable` and the number of arguments must match with the ones that the caller will pass to it.

When the user calls a client's function with a *handler* function, the client checks if the function is valid and saves it. After receiving the server's response, the client gets the user's function from the `pendingcommands dictionary` and then creates a `Deferred` instance with that function as a callback and executes it.

The `Deferred` class, also from the *Twisted* framework, is used as an execution synchronizing mechanism and acts as a proxy for a result that is initially unknown.

Since Python has not a direct implementation of events, the callback functions are defined using *setters* that also verify if the argument received is valid.

Unlike the C# client, this client establishes the connection to the *OSCD* server asynchronously which means that an *handler* function is also needed to this task. The same applies to the disconnection process.

4.3 *CardioWheel* Server

Since *CardioID*'s engineers had already developed a TCP server to communicate with the *BITalino* board, the connection to it and the management of the processing threads were already implemented and has a structure based on the one illustrated by Figure 4.23.

The server was implemented with *Python* and uses the event-driven networking engine *Twisted*, which has on its core a module called `reactor` that manages all events of the system.

The server is started by calling `reactor`'s function `run` which will enter in a loop and therefore, not execute the instructions declared after the call. To make the

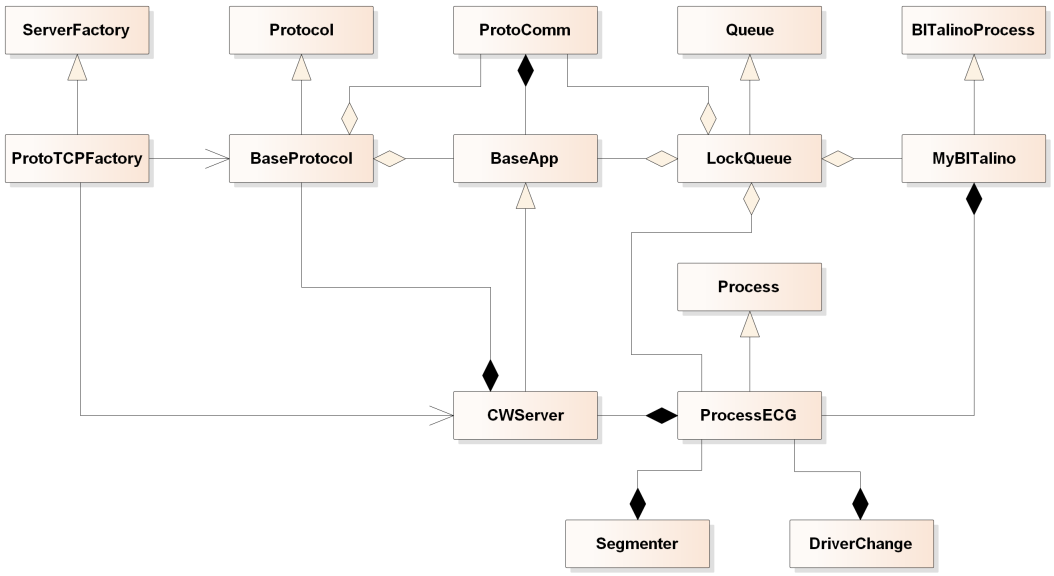


Figure 4.23: (Simplified) Structure of the *CardioWheel* Server (appx. A)

server listen for messages, a `ServerFactory` must be added to the engine. In this case, since the protocol used is based on TCP, the factory `ProtoTCPFactory` is added by using `reactor`'s function `listenTCP`.

After starting the execution of the `reactor` engine, the `ProtoTCPFactory` instance will be used to build the actual `Protocol` instance, in this case a `BaseProtocol` one, that will handle all the communication needed.

While waiting for connections, if a new one is made, then the protocol will use the class received as parameter on its constructor to build an instance of `BaseApp`, that in this case will be an instance of the `CWServer` class. The current instance of the protocol is used as parameter on the `CWServer`'s constructor. This process is illustrated on Figure 4.24.

After creating the `CWServer` instance its `_on_open` function is called, which is implemented on the `BaseApp` class. This function then unlocks the queue (instance of `LockQueue`) and calls the `on_open` function that is implemented by the `CWServer` class and that initializes the threads that communicate with the *BITalino* board and that process the data received from it.

Both the *CWServer* and the *BITalino* threads send information using the queue unlocked before. This queue is managed by the `LockQueue` class that, in a separated thread, checks if the queue has any message and sends them to the remove client through the `BaseProtocol` (Figure 4.25) instance.

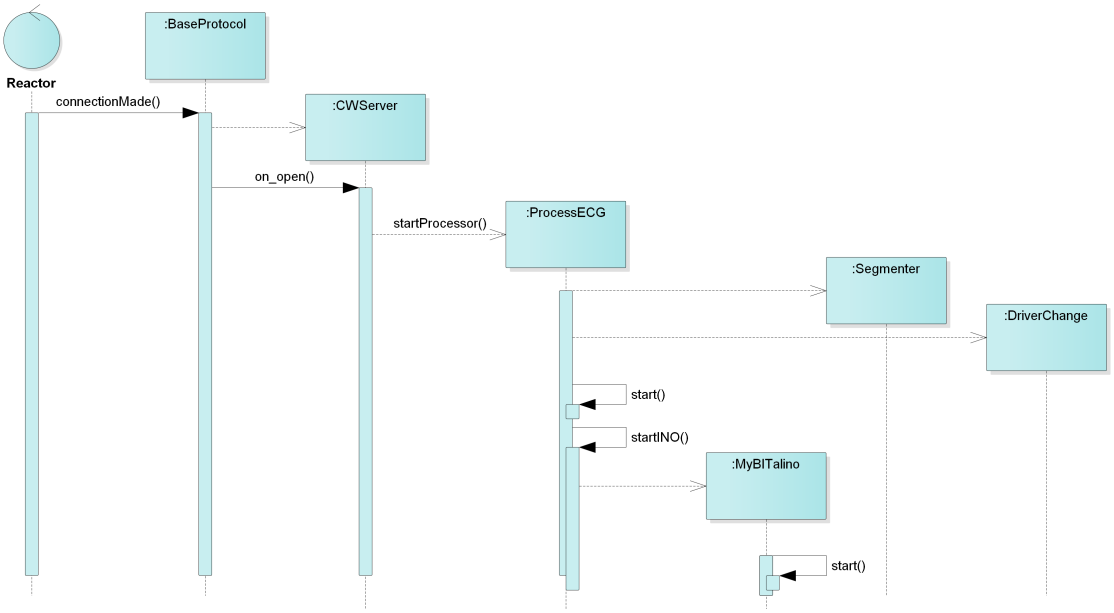


Figure 4.24: Sequence of actions executed when a connection is made with the server

4.3.1 Protocol Changes

Since the class `BaseProtocol` is from where the network messages are sent and received, it is only needed to modify the system at this point. The class `CWOSCDClient` (Figure 4.26) was created to make this adaptation and implements the function `send` that the original class also implemented.

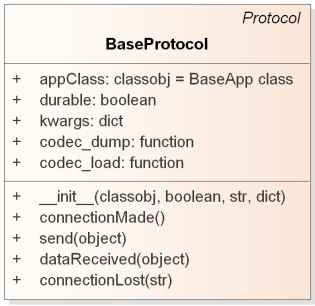


Figure 4.25: Class `BaseProtocol`

The `BaseProtocol`'s function `dataReceived` is called by the `reactor` module when data is received with the protocol. This function calls the `BaseApp`'s function `_callback` which can handle the data received. The data format used in the server is *JSON*.

In the same way, the `CWOSCDClient`'s functions `__on_start_sim` and `__on_stop_sim` receive *OSCD* publications and translate them to *JSON* so it

can be used by the `CWServer` class. To send data, the `CWServer` class continues to use the `send` function, which also receives data in *JSON* format and the `CWOSCDClient` class translates the received message to an *OSCD* publication.

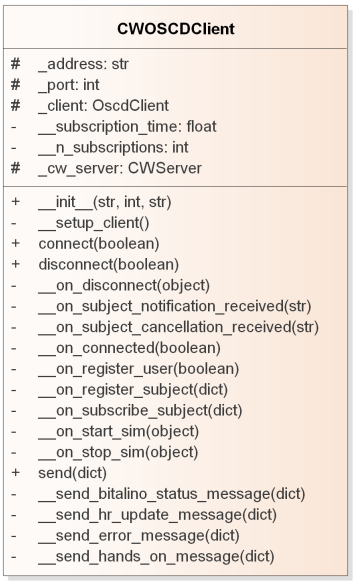


Figure 4.26: Class `CWOSCDClient`

The `CWOSCDClient` class also signals the `CWServer` one to open or close the connection, which will have a similar behaviour as the sequence illustrated in Figure 4.24, but is implemented in a way that only do that when all defined subjects are subscribed.

4.3.2 Behaviours and Messages

In order to be controllable, the `CWServer` class is listening for the following commands:

- `wait`:
  - Pauses the execution cycle of the *BITalino* threads;
  - Signals the device to stop acquiring;
- `acquire`:
  - Starts/resumes the execution cycle of the *BITalino* threads;
  - Signals the device to start acquiring;

- `NewPlayer`:
  - Starts/resumes the execution cycle of the *BITalino* threads;
  - Signals the device to start acquiring;
  - Starts a new acquisition session;
- `SimulationEnd`:
  - Starts/resumes the execution cycle of the *BITalino* threads;
  - Signals the device to start acquiring;
  - Stops the current acquisition session;
  - Processes the template of the current acquisition session;
  - Signals the remote client that the acquiring template as been processed.

While acquiring, the server can also send the following messages:

- `BITalino` - sent when tried to establish a connection with the *BITalino* board. It has a `boolean` attribute that indicates if it was successful;
- `UpdateHR` - sent when a new heart rate value was computed. This value is sent as a `integer` attribute of the message;
- `DriverChange` - sent whenever is detected that the user being tested it is not the same than before.;
- `Fatigue` - sent when is detected that user might be under fatigue;
- `Error` - sent when a client-relevant error has occurred. The message of the error is sent as a `string` attribute of the message;
- `HandsOn` - sent whenever is detected that the user's hands have been placed or removed from the sensors. It has a `boolean` attribute that indicates if the user has his hands on them;
- `TemplateReady` - sent when has completed the computation of the last user's heartbeat template. This includes the render of an image with it.

5

Driving Simulator

All the measures and modules described until this chapter will be acquired and used while the subjects are driving. As mentioned in section 1.2, it was decided that this project would be simulator-based for safety and ethical reasons.

The simulator's implementation was started by choosing a driving framework capable of providing a good experience to the subject. Being this module based on the *Unity 3D* framework, all the possibilities considered were available on *Unity*'s on-line store (*Asset Store*) and the one selected, due to monetary constraints, was part of the free *Standard Assets* [5] package of *Unity*. As expected, this solution is a good option for beginner-like projects but, in additions to the difficulty of tuning the components, its physics calculations are not accurate enough to give a good and realistic driving experience.

5.1 A Brief Look at *Unity* Framework

Before beginning to dive into the developed elements, it is important to understand how some aspects of the *Unity* engine work. There is two APIs[1] provided by *Unity*: `UnityEngine` and `UnityEditor`. The first one is used to script all the functionality of a game and auxiliary tools. The second API is used to develop components that will be included in the *Unity* editor, in other words, is

[1] Application Programming Interface

used to develop a user interface that helps to build games. The *Unity* framework also provides some management mechanisms to control the tools implemented by the game developer, such as the life cycle of a script (i.e. `MonoBehaviour` instance) and the event system implemented in the framework.

`UnityEngine` API

`GameObject` is the base class for all entities that exist in a Unity scene (or level) and each instance of this class can have several `MonoBehaviour` elements attached, which is a specialized implementation of the `Component` class that allows its inclusion in Unity's lifecycle execution, illustrated on figure 5.1.

As expected, the Unity framework has native components that can be added to `GameObject` elements, being the `Transform` component one that is always attached and visible, and provide the element's position, rotation and scale in the scene. To add physical behaviour, the framework also provides a `Rigidbody` and multiple `Collider` components, which respectively gives the element solid body physics and detect collisions with other `Collider` components.

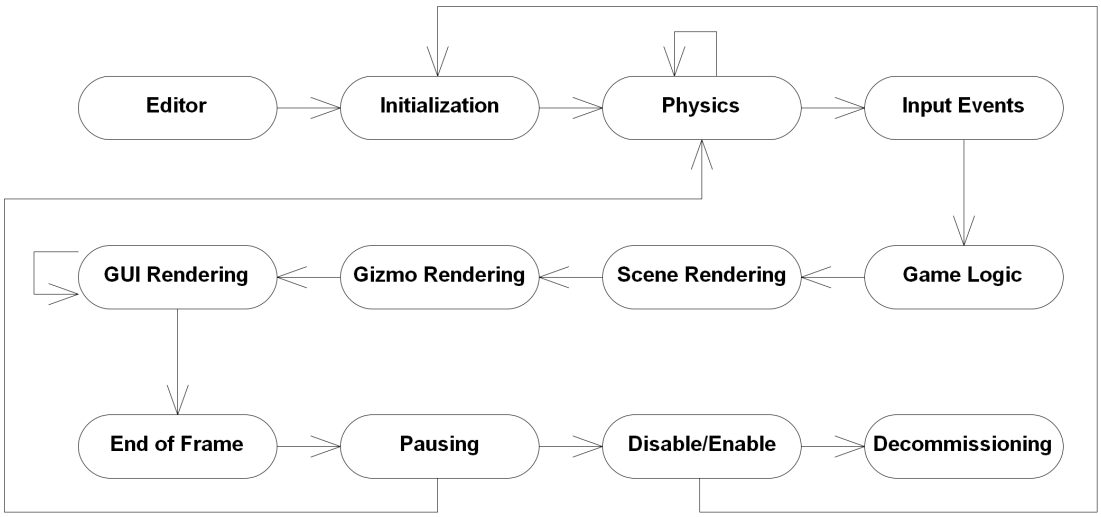


Figure 5.1: Low-detailed flowchart of the execution order of scripts in *Unity*

The *Unity* framework uses specially named functions that are called whenever some specific event occurs, being `Awake`, `Start` and `Update` the most common functions. The `Awake` and `Start` functions are called in the *Initialization* phase of the script's lifecycle and are usually used to do the work that otherwise would be done on the constructor of the class. On the other hand, the `Update` function is called in the *Game Logic* phase, executed every frame, and is usually where most

of the component's code will be executed. The *Physics* phase also has an update function named *FixedUpdate* that is called independently of the frame rate, which makes it the function to use for precise physics calculations.

`UnityEditor` API

It can be asserted that, in most cases, a certain functionality implemented with variable parameters will have, sooner or later, its values changed in order to tweak the results obtained. In some cases, the native *Unity* editor is enough to give the user a satisfying interface to changes these parameters but with more complex scripts, there is the need to add more information, to separate some parameters or to have a custom value manipulator.

The `UnityEditor` API allows the creation of custom windows, editors, property drawers and tree views. In addition to reducing the effort required to change certain parameters, the development of these custom interfaces allows the visualization of information in a more direct and intuitive way.

(a) Standard editor (b) Custom editor

Figure 5.2: `StressManager` component with and without a custom editor

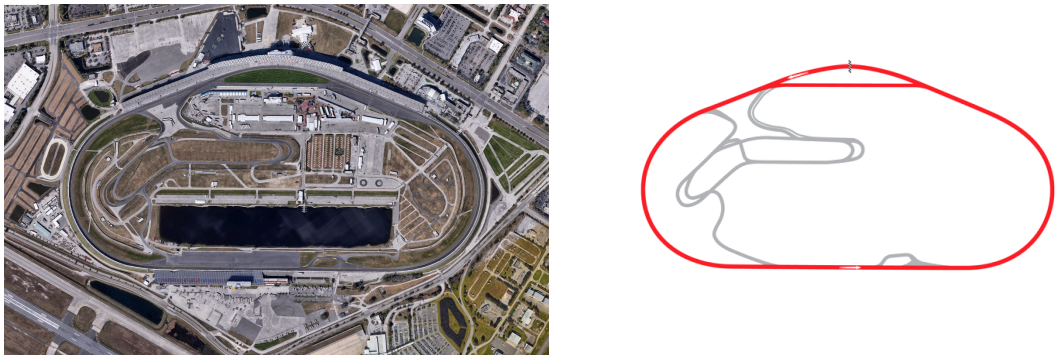
As shown on figure 5.2, the component `StressManager`[2] (not in use at the moment of the writing this report) has its own custom editor, that is used to arrange and show the information with more appealing designs. Unlike the standard editor that only lists the `StressInducers` components, the custom editor allows to manage the item's position on the list and shows it as a set of items with the

[2]Component developed to trigger actions based on the difference between the user's current BPM value and the BPM base line calculated at the begining of the session.

actual component, a slider to change its threshold of activation and a coloured rectangle that indicates if the inducer is enabled. The information related to the subject's BPM is also shown more like a control console, with sliders to change the calculation time, a button to start and stop calculating the BPM base line and a progress bar to give a visual feedback of the calculation process.

5.2 Simulation Environment

As a first step, the environment developed was design as a simple race track that, without opponents, can be monotonous enough to cause fatigue. This track was also designed in order to reduce the necessity of taking the hands out of the wheel or rubbing them on it. This precautions are necessary because the rubbing movement adds noise to the acquired ECG signal, which makes it more difficult to extract the required features. Taking the hands out of the wheel, even if just one of them, will obviously cause a disconnection between the system and the driver that interrupts acquisition of physiological signals.

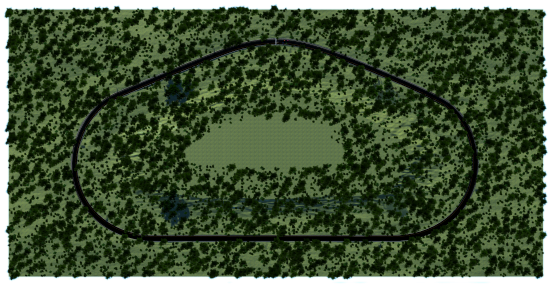


(a) Top-view (b) Tri-Oval Layout

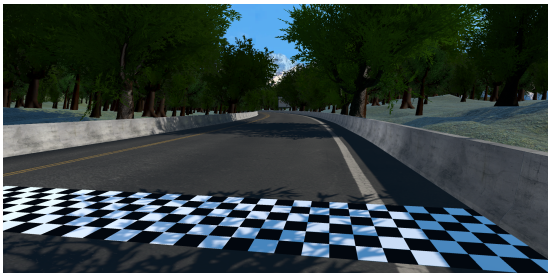
Figure 5.3: Daytona International Speedway

In order to reduce this effect, the selected track could not have many curves and the angle of these could not be too sharp. On a first version, using the iRacing track database (section 2.3), the *Daytona International Speedway*, with the Tri-Oval layout, was selected due to its open curves and simple layout without being only an oval shape track, illustrated on figure 5.3.

The track was modelled and textured using Blender [24], as well as all the other elements of the scene, such as the cement barriers that prevent the driver from leaving the asphalt. For better involvement, the track was surrounded by trees



(a) Modelled Environment (Top View)



(b) Modelled Environment (Prespective View)

(c) Environment Skybox

Figure 5.4: Simulation Environment

and the ground was raised in the center to better simulate a natural environment. For the distant landscape (i.e. *skybox*), the image in figure 5.4(c) was used.

All the materials of the scene's elements uses the computer graphics PBR[3] model, which enables a much more accurate rendering, independently from the environment where they are positioned. This rendering workflow can also achieve better photorealism by modelling the flow of light more accurately in comparison with the real world.

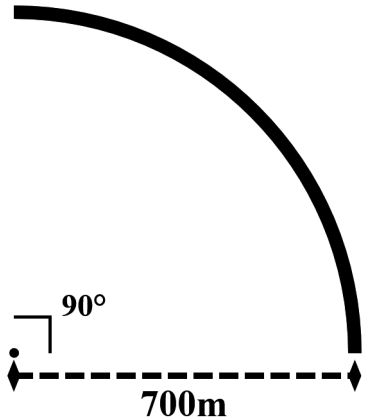


Figure 5.5: 90-degree curve minimum radius

Although the environment described above was already designed for a smooth

[3]Physically Based Rendering

driving experience, the results revealed that the track were still very challenging, especially in high speeds. For this reason, a new simulation environment was built and, to avoid this kind of shortcomings, this new track was design using real scale models and taking in consideration the highway Portuguese standards, which defines that, as illustrated on figure 5.5, for a speed limit of 120 kilometers per hour, a 90-degree curve should have a minimum radius of 700 meters [13].

Figure 5.6: Road block (5-degree curve)

Another major change relatively to the first version of the track was the design of "road blocks" (illustrated on figure 5.6) rather than an entire and complete track. This different approach, besides giving an relatively small increase in performance, allows the customization of the track and the design of several tracks using the same elements.

At the moment, there are three two lane highway elements fully modelled and textured: a straight 40-meter block and a left and right version of a 5-degree curve.

Finally, to give the simulation scene a more realistic visual appeal, the environment was based on heightmaps[4] captured from the real-world, at a scale of 1:10 (one meter to ten meters).

The first track was design as a 3 by 3.4 kilometer rectangle in a forest like environment. These dimensions make the driving experience more monotonous which makes it much more likely to induce fatigue. The simulation environment is also improved by using a parallel road for the incoming traffic (illustrated on figure 5.7(b)).

`Road Builder` Extension

The process of building a *n*-degree curve using blocks with 5 degrees can be very difficult and time consuming due to the precision needed while positioning them, as the coordinates of each block can have long decimal places that must be exactly

[4]Grayscale image where its values are interpreted as displace distance (black is the minimum height and white is the maximum)

(a) Top View (b) Perspective View

Figure 5.7: Second Simulation Environment

introduced. Using the figure 5.8 as an illustration of an arch of the unit circle, and knowing that $\angle AOB = 5°$, it is calculated that the coordinates of point B will be $\approx (0.996, 0.087)$.

$$A = (1, 0)$$

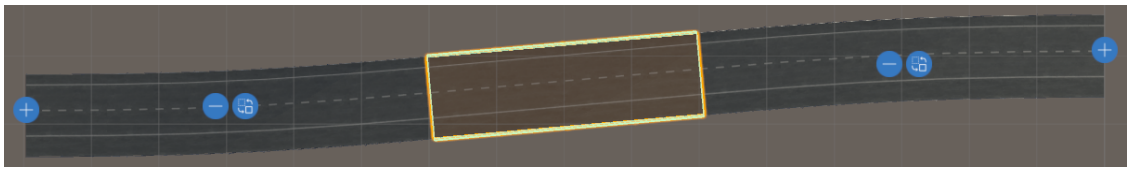
$$B = (\cos(5), \sin(5)) \approx (0.9961734827, 0.0873978972)$$

Figure 5.8: 5-degree arch from unit circle

To build a 3-kilometer track using 40-meter blocks and position them with great precision, would be a very difficult and tedious process. The "*Road Builder*" *Unity* extension was created to automate this process.

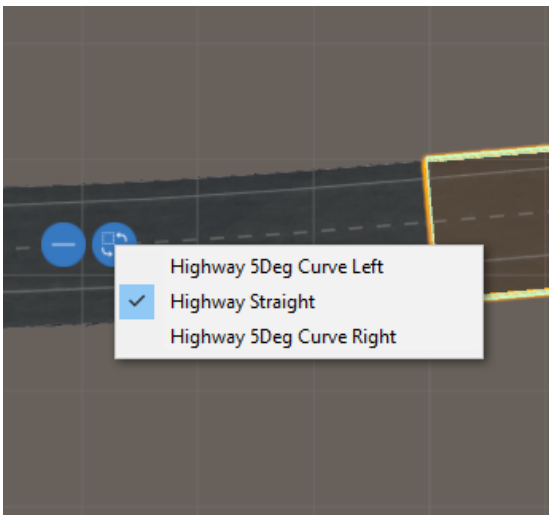
As illustrated on figure 5.9(a), the "*Road Builder*" extension provides three handles for all blocks that are ends of a road: *Add*, *Remove* and *Change*. The last handle allows to change the current block type to any of the ones available.

After building and positioning the track in the simulation environment, it is necessary to change the height of the terrain so it adapts to the vertical position of the track. As previously mentioned, in *Unity*, a terrain element gets its displacement (i.e. height) from a grayscale 2D texture, illustrated on figure 5.10(a), and any change on those values reflects as 3D modifications on the terrain mesh.



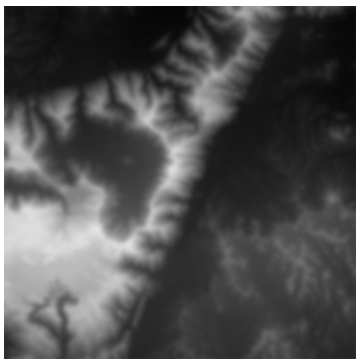
(a) Scene UI

(b) Main Window

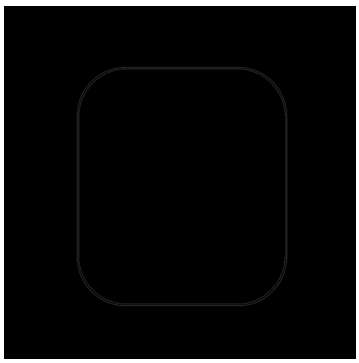


(c) Change Menu

Figure 5.9: Road Builder Extension - Graphical Interface



(a) Original



(b) Ray Cast

(c) Result

Figure 5.10: Road Builder Extension - Heightmap Processing

The first step was to iterate every point equivalent to a pixel on the heightmap and to ray cast[5] from the minimum to maximum height. If any road block intercepts the ray, its vertical coordinate is registered on temporary "texture", illustrated on figure 5.10(b). Hereinafter, these temporary textures will be referred to as matrix, since a 2D texture is nothing more than a 2D matrix.

[5]Ray tracing technique that shoots one ray per pixel, and find the closest object blocking the path of that ray.

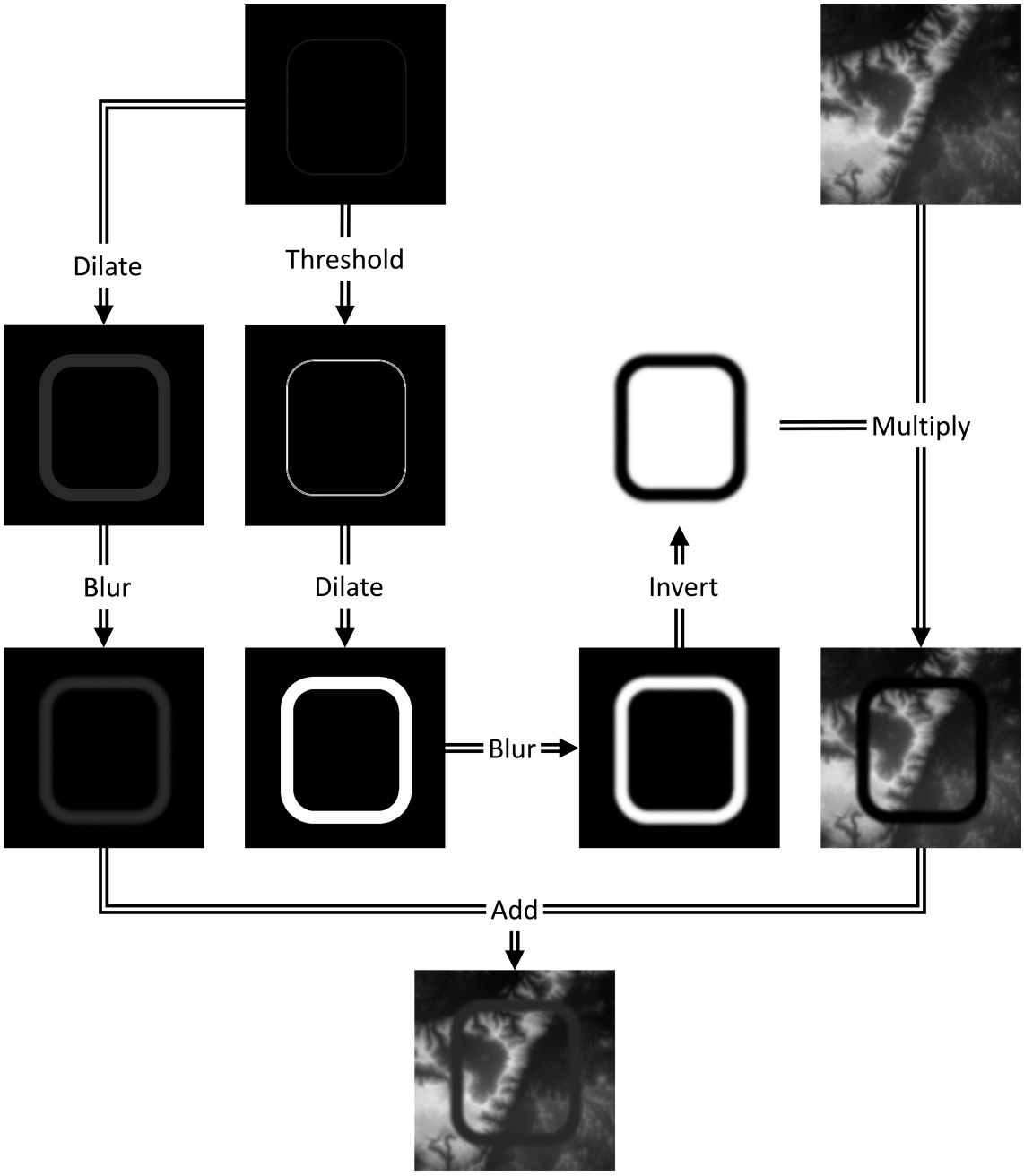


Figure 5.11: Road Builder Extension - Heightmap Processing (Steps)

After the ray casting process, and by using image processing, the result matrix is then smoothly "merged" with the original data in order to get the final heightmap texture, illustrated on figure 5.10(c).

The image processing applied, illustrated on figure 5.11, begins with a binary filter, with the threshold value at `0`, applied to the ray cast matrix. This means that any value greater than `0` it is considered to be `1`, otherwise is registered as

`0`. The result of this filter, also known as a mask, is then dilated[6] using an ellipse structuring element, which "expands" the threshold result and also removes the gaps between the lanes.

After the last dilation, it is used a Gaussian blur filter to smooth the dilated matrix edges that is then inverted so the `1`'s become `0`'s and vice versa. The terrain original heightmap will be multiplied by the result of these two operations so it removes the height data that will be replaced by the heights of the lanes.

This height data of the lanes, that was obtain on the ray cast process described before, also need to be dilated and blured, with the same values of the previous steps, to match the mask that was used to remove data from the original heightmap. Finally, these two height data matrix are added to make a complete version of the terrain heightmap with the lane heights carved on it, as illustrated on figure 5.12.

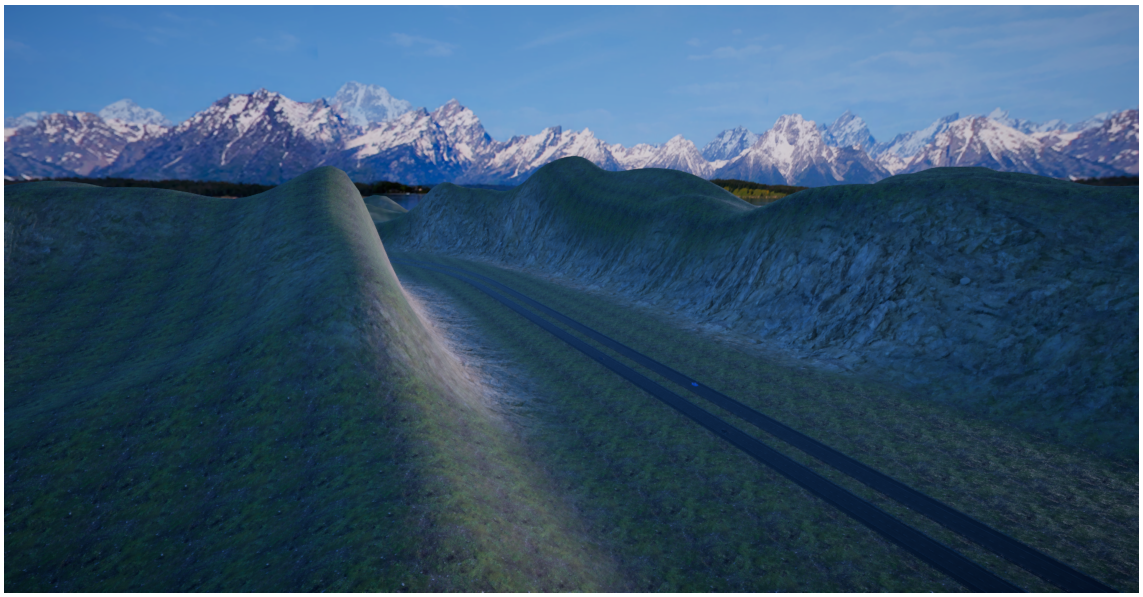


Figure 5.12: Road Builder Extension - Heightmap Processing (Result)

Another important part of this nature environment, in the sense that makes it more believable, is the inclusion of trees. *Unity* terrain has an automatic tree spawner that randomly places a given number of tree instances on the available area, while also having in attention the terrain inclination. Unfortunately, this means that many of the trees will be intercepting the track and need to be removed or, optimally, repositioned.

[6]One of the basic operations in mathematical morphology and usually uses a structuring element that indicates how the process is applied to the input image.

In order to automatize this process, it was implemented on the `Road Builder` extension a procedure that iterates all the existent tree instances and ray casts from its position to the maximum height. If the casted ray intercepts a road block, the correspondent tree instance is removed.

Finally, it was also implemented on the extension a procedure that extracts the *waypoints* references from the road blocks and groups them per lane, process that has been implemented taking into account that the road blocks include these references.

The term *waypoint* comes from the use of *Unity* "Standard Assets" package, which has two AI navigation aid components: `WaypointCircuit` and `WaypointProgessTracker`. A `WaypointCircuit` component groups registered references as points of a path and interpolates between them. This path can them be accessed to get a position in it, usually through the usage of the `WaypointProgessTracker` component.

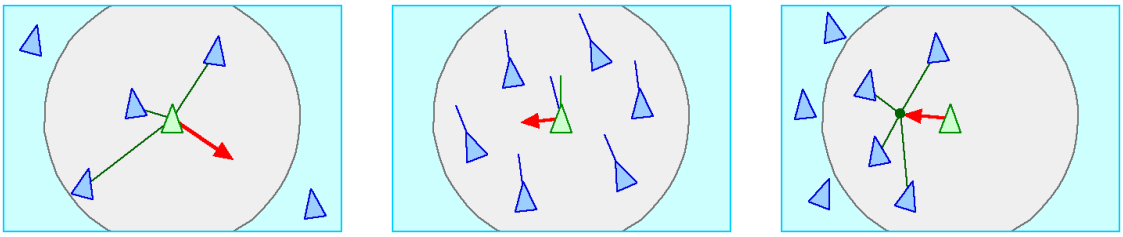
After this whole process, there should be a track seamlessly integrated with a forest-like terrain and with elements to be used in AI navigation.

5.3 Artificial Intelligence

As an approach to introduce "traffic" in the simulation, it was developed a package named *Boids* that implements a model later classified as swarm intelligence, a subset of the artificial intelligence field [29] [62]. This give to an autonomous agent the capacity to simulate the flocking behaviour of birds, which is the combination of 3 separate rules: separation, alignment and cohesion, all illustrated on figures 5.13. The first rule steer the *Boids* to separate from each other, avoiding local crowding and overlaps. The alignment rule adds a force that steers the agent toward the average heading of its neighbours. The third rule - cohesion - forces the agent to move toward the average position of the group, its center of mass.

This behaviour does not seems to be suitable for the task of steering a group of virtual cars but it is possible to give the agent a target to reach or a path to follow by adding more rules and weighing their influence on its final behaviour. It can also be added randomness, also known as wander, to the acceleration and steering forces of the agent to give it a more realistic and natural movement [66].

All these properties related to the *Boid* agent as a single element are stored on the `BoidProperties` class, which includes its maximum speed, maximum force



(a) Separation Rule (b) Alignment Rule (c) Cohesion Rule

Figure 5.13: Basic Rules of the *Boids* Model [62]

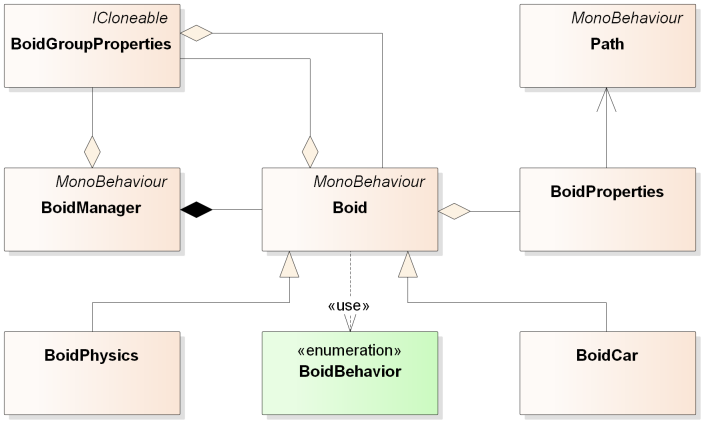
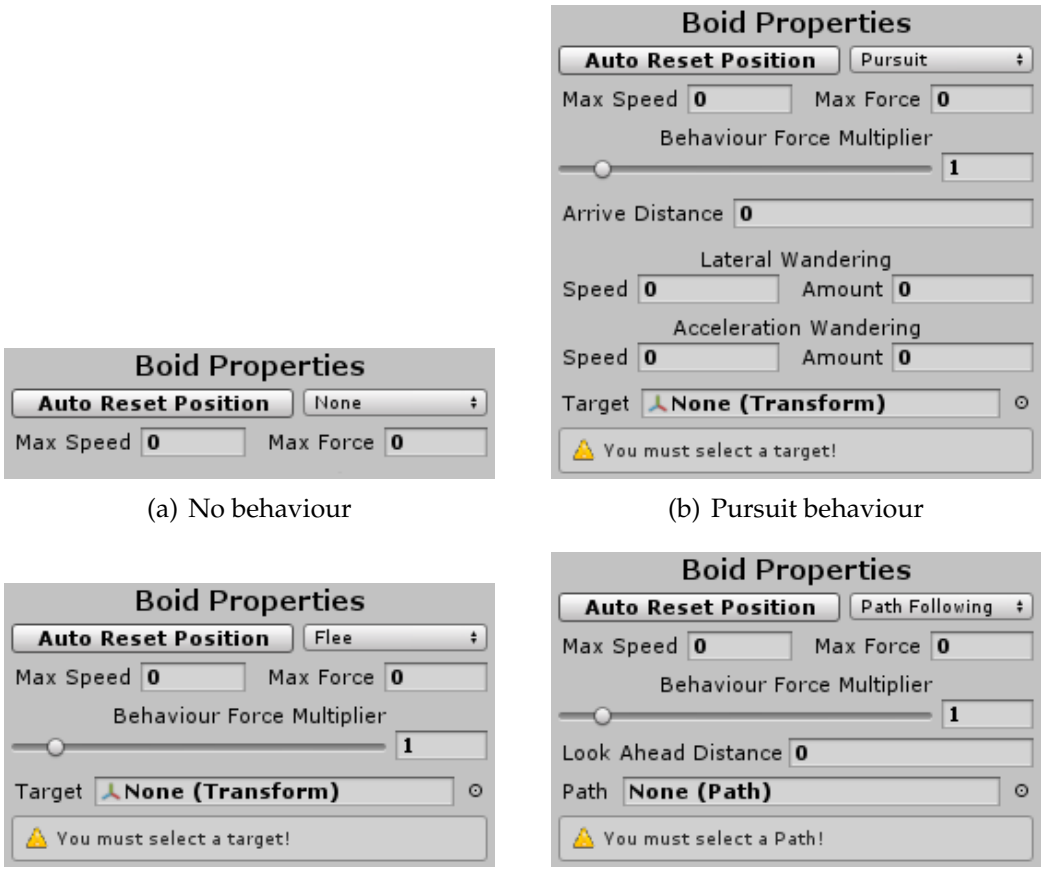


Figure 5.14: (Simplified) `Boids` package (appx. E)

and look ahead distance, three of the parameters that have major influence on the agent's performance. On the other hand, the `BoidGroupProperties` class stores the parameters that enables the coordinated movement of all *Boid* agents. Both the `BoidProperties` and `BoidGroupProperties` classes are used by the `Boid` and the `BoidManager` classes and have a custom drawers to be shown on the editor with a more intuitive design.

The current implementation of the *Boids* theory allows 3 possible individual behaviours: pursuit a target, flee from one or follow a `Path` class instance. These parameters are store in the `BoidProperties` class and the custom property drawer of this class, figure 5.15 uses the `BoidBehavior` enumeration to only show the parameters that are related to the selected behaviour, in addition to the common `maxSpeed` and `maxForce` parameters. All the behaviours have a force multiplier to boost or attenuate its influence on *Boid*'s action and both the pursuit and flee behaviours have a target that is used to calculate the desired velocity. When in pursuit mode, there are also shown values for the arrive distance, lateral and acceleration wandering, which give the agent a more natural behaviour when moving and arriving the destination.

When in path following mode, the *Boid* uses an instance of the `Path` class to get the position to where it should be targeting. This obtained target can be positively offset so the agent can "predict" where the target is going to be and adjust its velocity accordingly to that information. If set correctly, this parameter can make the agent follow the given path in a very smoothly way but it can be easily overcome.



(a) No behaviour

(b) Pursuit behaviour

(c) Flee behaviour

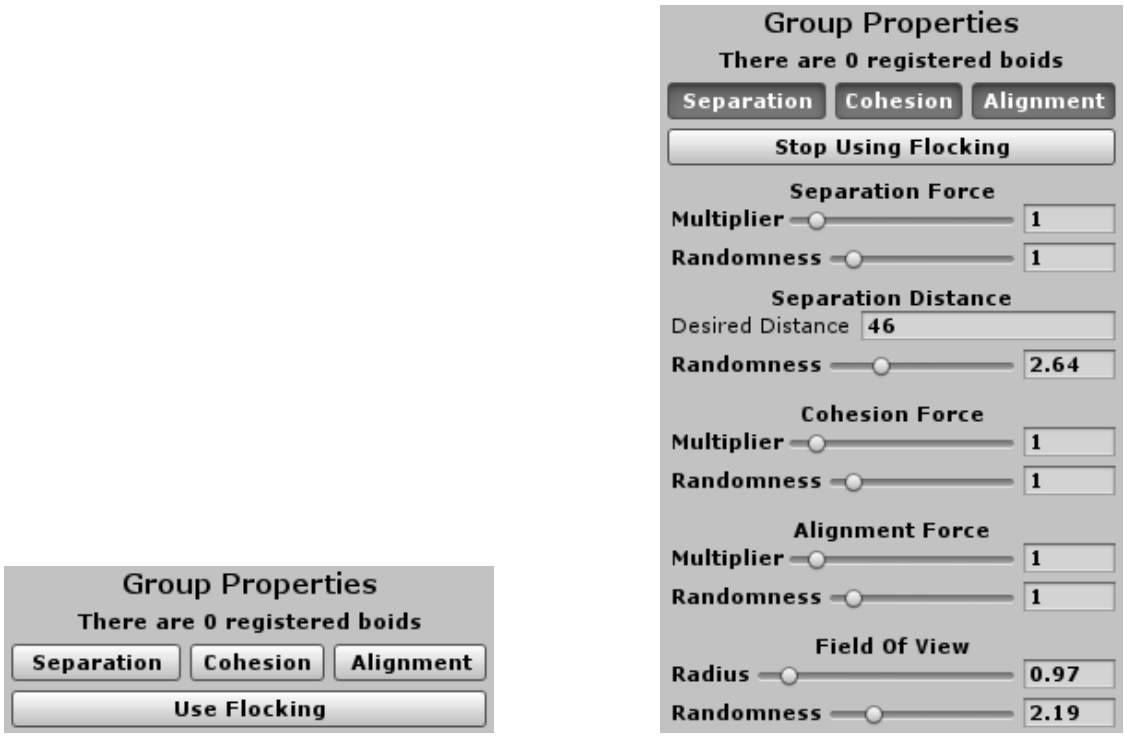
(d) Path following behaviour

Figure 5.15: `BoidProperties` property custom drawer

As mention before, in addition to the `BoidProperties` class, which stores data that is used to compute the individual behaviour of a given agent, there is the `BoidGroupProperties` class that provides information to that same *Boid* about is neighbour agents. This class also has a custom drawer, illustrated on figure 5.16 that adds four buttons to the editor: "Separation", "Cohesion", "Alignment" and "Use Flocking" or "Stop Using Flocking". As mentioned before, *flocking* describes the behaviour of a group of birds while in flight and so, the last button is simply a shortcut to, respectively, activate and deactivate all the group behaviours.

All the group behaviours have a multiplier and randomness factor that are used

to scale its influence on the final force that will be applied to the agent. The randomness factor add some variation to the *Boid* agents that make a given group. When using the cohesion or alignment behaviours, it is also shown two parameters that manipulate the field of view radius for each *Boid* element. If the separation behaviour is activated, two more parameters are shown that specifies the desired distance between the agents that make up the group.

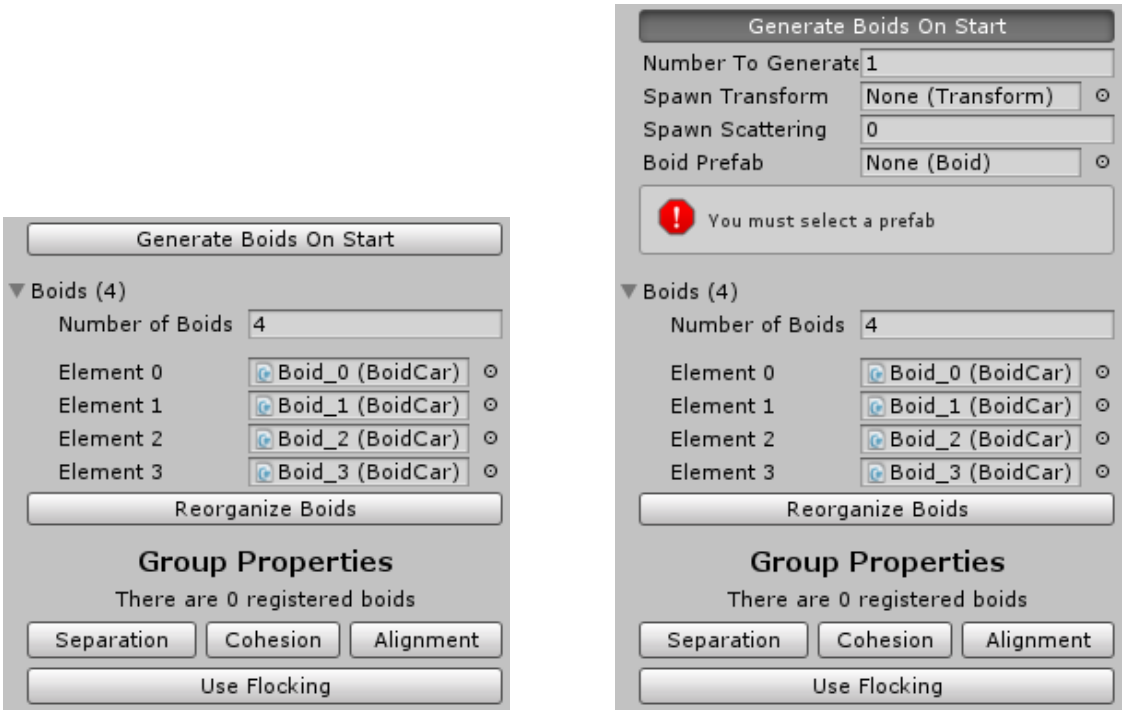


(a) No group behaviour (b) All behaviours

Figure 5.16: `BoidGroupProperties` property custom drawer

It is important to note that the `BoidManager` class does not provide any guidance to instances of the `Boid` class because, as autonomous agents, they are responsible for their own "choices" in how to act in the environment that surrounds them. With this in mind, the `BoidManager` class is responsible for keep a list of all the *Boid* that should interact with each other and update their group properties. It can also generate, at the beginning of the level, instances of a given `Boid` prefab at a given position and, by using the "Reorganize Boids" button on its custom editor, also allows change the name and parent of the `Boid` instances associated to it. The group properties are shown using the custom property drawer mentioned before.

The *Boid* model implementation starts by loading the position and rotation of the element on the scene. Every frame, the *Boid* agent uses its properties to calculate



(a) Default (b) With Automatic *Boid* Generation

Figure 5.17: Custom editor of the `BoidManager` component

the force vector that will steer it and sums that force to its velocity vector, that later updates its position. This sum between the calculated force and the velocity of the *Boid* cannot be done directly, being necessary the usage temporal discretization so it can be applied throughout the frames evolution.

The forces that simulate the swarm behaviour, as suggested in the *Boids* theory of Craig Reynolds, are calculated using all the *Boid* neighbours using a similar process, as described in listings 5.1, 5.2 and 5.3. This obviously means that the listed code is highly optimizable.

All the three functions uses an auxiliary function named `Steer` that receives the desired velocity of the *Boid*. This "desired velocity" is a vector begins at the agent's position and ends on the target, which means that this vector could have an unlimited value of magnitude and, therefore, a unlimited velocity. To prevent this effect, the `Steer` function normalizes the input vector and gives it a new magnitude based on the `maxSpeed` property. Using this clipped desired velocity, it is calculated the steering vector that should be added to the acceleration of the *Boid*. This steering vector is the difference between the current velocity and the desired one and it is also limited by the `maxForce` property.

```
public Vector3 GetSeparationForce() {
    var separationVector = Vector3.zero;
    var nBoids = 0;
    foreach (var neighbour in groupProperties.neighbours) {
        if (neighbour == this) continue;
        var diffVector = transform.position - neighbour.transform.position;
        var distance = diffVector.magnitude;
        if (distance < groupProperties.desiredSeparation) {
            separationVector += diffVector.normalized / distance;
            nBoids++;
        }
    }
    if (nBoids > 0) return Steer(separationVector / nBoids);
    else return Vector3.zero;
}
```

Listing 5.1: Separation force calculation function

```
public Vector3 GetAlignmentForce() {
    var alignmentVector = Vector3.zero;
    var nBoids = 0;
    foreach (var neighbour in groupProperties.neighbours) {
        if (neighbour == this) continue;
        var distance = (transform.position - neighbour.transform.position).magnitude;
        if (distance < groupProperties.viewRadius) {
            alignmentVector += neighbor.velocity;
            nBoids++;
        }
    }
    if (nBoids > 0) return Steer(alignmentVector / nBoids);
    else return Vector3.zero;
}
```

Listing 5.2: Alignment force calculation function

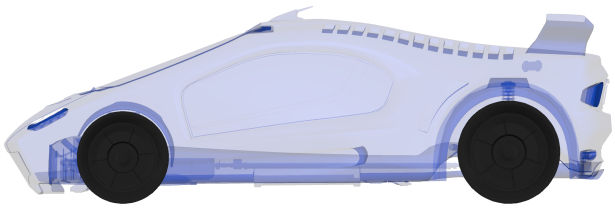
```
public Vector3 GetCohesionForce() {
    var cohesionCenter = Vector3.zero;
    var nBoids = 0;
    foreach (var neighbour in groupProperties.neighbours) {
        if (neighbour == this) continue;
        var distance = (transform.position - neighbour.transform.position).magnitude;
        if (distance < groupProperties.viewRadius) {
            cohesionCenter += neighbour.transform.position;
            nBoids++;
        }
    }
    if (nBoids > 0) return Steer(cohesionCenter / nBoids - transform.position);
    else return Vector3.zero;
}
```

Listing 5.3: Cohesion force calculation function

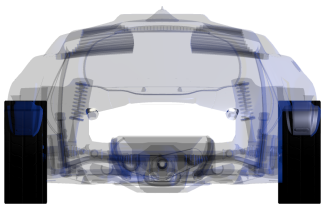
```
protected Vector3 Steer(Vector3 desiredVelocity, bool slowdownArrival = false)
{
    var dMag = desiredVelocity.magnitude;
    var steerForce = Vector3.zero;
    if (dMag > 0) {
        desiredVelocity = desiredVelocity.normalized;
        if (slowdownArrival && properties.arriveRadius != 0 && dMag < properties.
            arriveRadius)
            desiredVelocity *= dMag.RemapUnclamped(0, properties.arriveRadius, 0,
                properties.maxSpeed);
        else
            desiredVelocity *= properties.maxSpeed;
        steerForce = desiredVelocity - velocity;
        if (steerForce.magnitude > properties.maxForce)
            steerForce = steerForce.normalized * properties.maxForce;
    }
    return steerForce;
}
```

Listing 5.4: Steer function

(a) Perspective view



(b) Side orthographic view



(c) Front orthographic view

Figure 5.18: Semi-transparent *boids*

With this steering process in mind, the behaviour of following a given target is easily implemented using the distance between the target and *Boid* position as

input of the *Steer* function. For a smooth arrival, this function can be flagged to reduce the steering force according with the distance to the target.

All this behaviour is applied to the car controller component in an specialized class named `BoidCar`. Because this controller component uses float values between $[-1, 1]$ to accelerate and steer the object, this class executes the previously mentioned calculation on the `FixedUpdate` event function and translates its results to this interval. As an attempt to make *Boids* aware of the player, a collision avoidance system based on raycasts was implemented to produce a new force vector that would be summed with the steering force calculated previously.

NOTE: The result of this implementation was too volatile to be used on the simulation. For this reason, in this version of the simulator, the player was moved to a different collision layer than the *Boids* so they could not collide with it. The driver perceives this "solution" as an semitransparent *Boid* rendering, figure 5.18.

Figure 5.19: *Boid* 3D models

As mentioned before on section 5.2, it was design a new track to fix some of the problems that the first version had. The same happened with the *boids*, not implementation-wise but on their properties. Each *Boid* element was configured to be in `Pursuit` mode with its target also being referenced in a `WaypointProgressTracker` component, which updates the target position based on a given `WaypointCircuit` instance, as mentioned also on section 5.2.

As a group, there *Boid* elements were in flocking mode (i.e. with separation, cohesion and alignment forces active) with the separation force multiplier 1.5 times

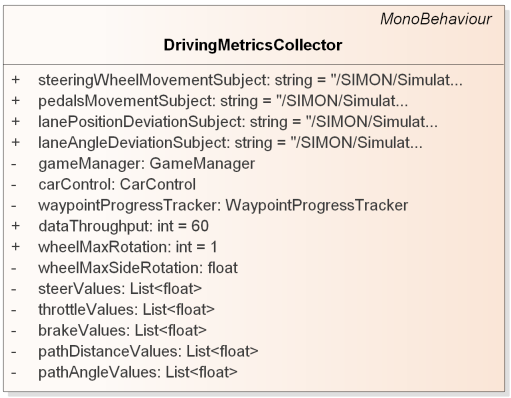
stronger than the others, the desired distance at 10 meters and a field of view radius of 2 meters. These parameters made the *Boid* agents consistent enough to drive besides them.

Since there were now four different lanes on the scene, it was assign a `BoidManager` component per lane so their *Boid* agents did not take in consideration the ones driving on the other lanes. To spawn these agents, it was created a component that randomly instantiate a given number of pre-selected elements at the beginning of scene. Visually, the agents were also updated with more realistic 3D models, as illustrated on figure 5.19.

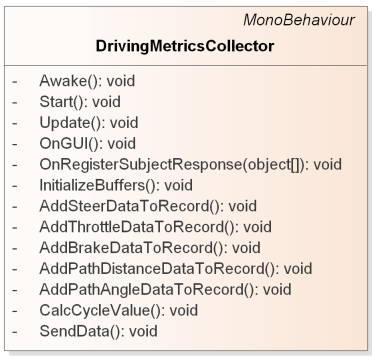
The *Boid* component package was also used to drive the player's car during the first 5 seconds of the simulation because, since the experiment takes place in a highway environment, it was not logical to start it with the vehicle stationary on the middle of the road.

5.4 Performance Measuring

In section 2.2.3 is was described some measures usually used to evaluate the driver's performance. The `DrivingMetricsCollector` class, illustrated on figure 5.20, has added to the player `GameObject` to given it this capability and does it by using one list per type of value and filling them in every frame. This values are computed and published `dataThroughput` times a second, followed by the complete clearing of the buffering lists. This process is schedule using the *Unity* `InvokeRepeating` function with a repeat frequency of $^{1}/_{dataThroughput}$ Hz.



(a) Attributes



(b) Operations

Figure 5.20: `DrivingMetricsCollector` class

Since it is unlikely that there would be only one sample available per computing cycle, it is necessary to have a descriptor of the collected data. As expected, this descriptor is highly dependent of the `dataThroughput` value, which means that as its value decreases so does the precision of the descriptor. Since the simulator updates its state several times per second, it also acquires performance data at that same rate. This rate is not static or predefined, which means that the amount of data used to calculate the measures descriptors is not specified.

Table 5.1: Descriptors used for each performance measure

Performance Measure	*Descriptor*
Steering Wheel Movement	Mean + Standard Deviation
Throttle Movement	Derivative
Brake Movement	Derivative
Lane Position Deviation	Mean + Standard Deviation
Lane Angle Deviation	Mean + Standard Deviation
Speed	Mean + Standard Deviation
Collisions	Object + Force
Lane Change	Name

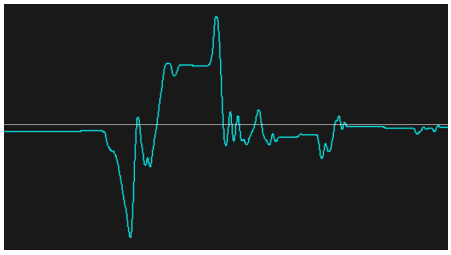
While with SWM[7] and lane position and angle deviation the actual value collected can be useful to extract tendencies (e.g. number of steering wheel zero-cross occurrences), the movement of pedals does not have such useful information in its raw values but rather in its change velocity. For this reason, both the throttle and brake movement measures are described by their derivative.

After acquiring data using the first simulator solution, which only included SWM, throttle and brake movement, and lane position and angle deviation, there was a lack of information about the "state" of the simulation. In an attempt to fill the information gap, the second version of the simulator also included the player's car speed and events triggered by lane changes and collisions with both the environment and other cars. These lane changed and collision events are sent immediately after the event occurs on the simulator.

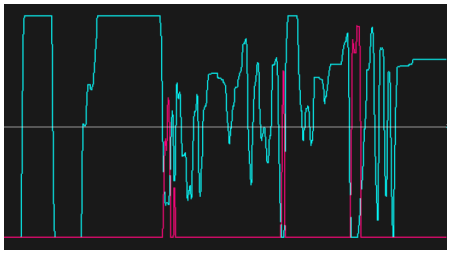
5.5 OSCD Integration

As expected, this module will also need to be connected to the *OSCD* communication architecture. At the moment, there are two *OSCD* clients in this module:

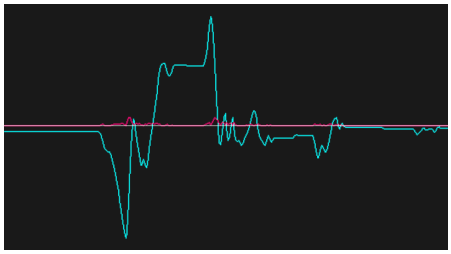
[7] Steering Wheel Movement



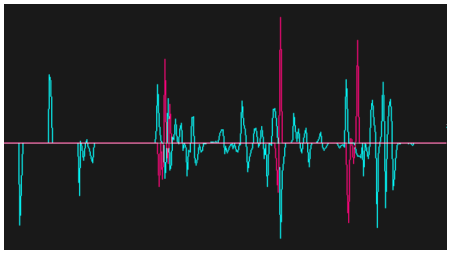
(a) SWM raw data



(b) Throttle (cyan) and brake (pink) movement raw data



(c) SWM mean (cyan) and standard deviation (pink) descriptors



(d) Throttle (cyan) and brake (pink) movement derivative descriptors

Figure 5.21: Wheel and pedals raw data versus their descriptors

one for the exchange data with the *CardioWheel* system and another to publish the collected measurements. This task could obviously be done with only one client but these solutions were implemented in different occasions and the refactorization of code was not viable due to defined time deadlines.

The *CardioWheel*-related *OSCD* client is encapsulated in a package called `BITalinoClient` (previous name convention), illustrated by figure 5.22. The `BITalinoClient` class, being a specification of *Unity* `MonoBehaviour` class, can be attached to `GameObject` instances and exposes event handlers that are invoke when the *CardioWheel* module publishes subscribed data, which includes heart rate updates, fatigue and driver change detection. This class also allows the publication of the signals `StartingSimulation` and `StoppingSimulation`.

To be able to intuitively change the component's properties and manage the client's connection to the server, it was created a custom editor for the `BITalinoClient` component, illustrated on figure 5.23, that gives visual feedback of the connection state, as well as a message log with all the actions and its results performed by the client. Conveniently, it is also used to give users the ability of folding all the available event handlers into just a label.

Because there are multiple scenes with multiple elements that might need to interact in some way with the simulator state, a specification of the `Component` class, named `GameManager`, has been implemented as a static attribute in the sense

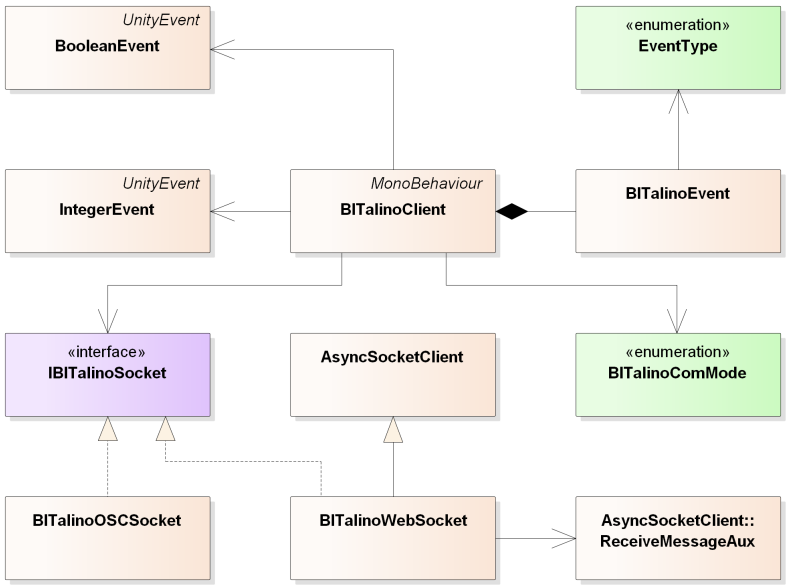
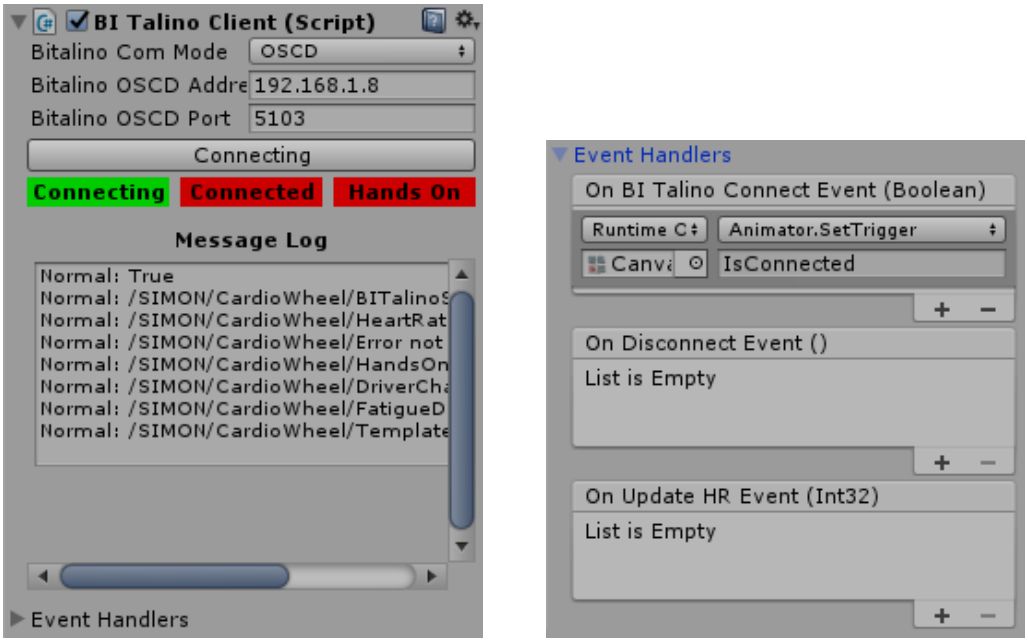


Figure 5.22: (Simplified) `BITalinoClient` package (appx. F)



(a) Connection properties (b) Some of the event handlers

Figure 5.23: Custom editor of the `BITalinoClient` component

that it is accessible by all the other `Component` elements and it is not destroyed when switching scenes, something that happens to all the others elements. This class keeps track of all the settings of the simulator and exposes an instances of `BITalinoClient` and `OscdClient`, the two *OSCD* clients mentioned before. For this reason, this class is crucial for the interconnection of the system and,

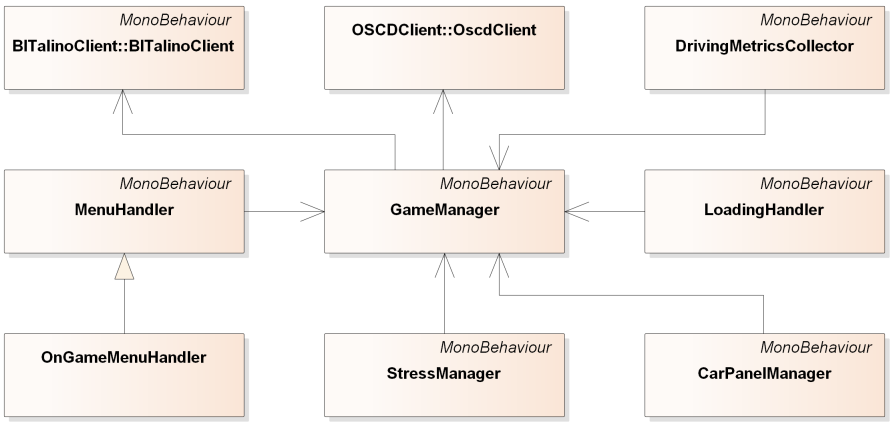


Figure 5.24: Overview of the classes associated to `GameObject` (appx. G)

as illustrated in figure 5.24, is associated to several elements. By exposing the `BITalinoClient` instance it enables the other elements to add event listeners so they can react to *CardioWheel* events.

On previous section has already been mentioned that the simulator also publishes subjects. The addresses of these subjects are leafs of the address branch `/SIMON/Simulator` and those leafs are:

- `/StartingSimulation` - Published after loading the track scene;
- `/StoppingSimulation` - Published when the simulation is stopped;
- `/SteeringWheelMovement` - Published at the specified frequency (section 5.4) with the mean and standard deviation of the SWM variations;
- `/PedalsMovement` - Published at the specified frequency with the derivatives of the throttle and brake pedals movement;
- `/LanePositionDeviation` - Published at the specified frequency with the mean and standard deviation of the distance of the player to the "optimal" path (figure 5.25);
- `/LaneAngleDeviation` - Published at the specified frequency with the mean and standard deviation of the angle between the player's direction and the "optimal" path;
- `/CurrentVelocity` - Published at the specified frequency with the mean and standard deviation of the speed of the player;

- `/CollisionDetected` - Published every time the player collides with something. Each publication includes the force of impact and name of the object that the player collided with;
- `/LaneChanged` - Published every time that the player changes lane and includes its name.

Figure 5.25: Path (bright yellow) used as the optimal one

The majority of the subscribed subjects from the *CardioWheel* module triggers events that are shown on the car front panel during the simulation.

5.6 GUI and Scene Management

Before introducing the GUI[8] it is important to mention that some elements of the interface were design for the first version of the simulator, which was design to induce and test fatigue while driving. This project was develop as a *CardioID* tool using the *CardioWheel* device.

Until this moment, the simulator consists of 3 scenes: the connecting screen, the main menu and the simulation environment. The first scene initializes the `GameManager` component and makes it available for all the scenes, which means that must always be the scene to start the system. While attempting to connect to the *OSCD* server and receive confirmation of connection to the device from the

[8] Graphical User Interface

CardioWheel module, both the text and the heart figure fade in loop with a breath-like movement. If succeed, the text fades-out and the heart figure shrinks and if the *SPACE* key was pressed, the connection attempt is skipped and the screen behaves as it succeed.

(a) Connect phase (b) Skip phase

Figure 5.26: Connecting screen

The connection screen fades-out to the main menu. Here, the user can start the simulation, change some minor settings and disconnect, or connect to the *OSCD* server. In the bottom left corner of the menu canvas there is a chip symbol that indicates either the simulator is connected (white) or disconnected (black) from the server.

When the "*START*" button is pressed the scene fades-out and the simulation environment scene starts. In this scene, the camera is placed inside the car that the subject will drive and a few important aspects are visible on the front panel of the car: a heart BPM[9] and current velocity indication as well as a heart figure. This figure has a looping zoom animation that is triggered by the `HandsOn` event from the *CardioWheel* module, which means that if the driver has both his hands on the steering wheel the heart figure starts to pulse, being static otherwise.

If the *CardioWheel* module detects a new driver on handling the steering wheel, it publishes with `DriverChange` subject that, in the simulator, triggers a *welcome* notification that appears on the car's panel, figure 5.29(a). In a similar way, if fatigue is detected on the driver's physiological signals, an alert notification shows

[9] Beats Per Minute

(a) Main page (b) Settings page

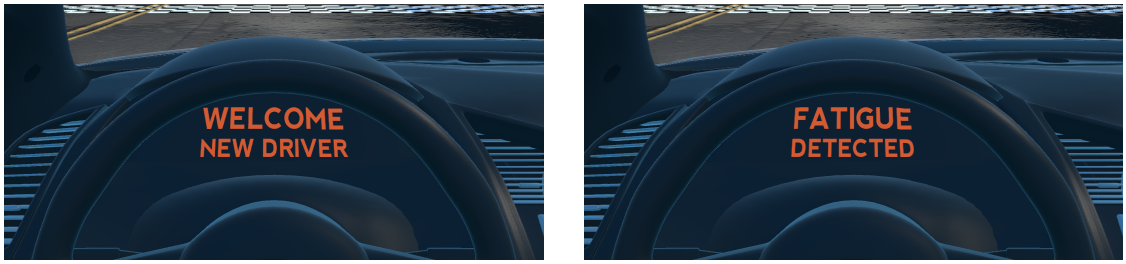
Figure 5.27: Main menu screen

Figure 5.28: Main menu screen

on the panel, as illustrated by figure 5.29(b).

On the second version of the simulator, it was added to the car panel a stopwatch that counts the elapsed time since the beginning of the simulation. It was also implemented the possibility of iterate over multiple cameras, including one on the hood to later be used as input of a *MobilEye* device, as mentioned on section 2.2.3.

Another feature implemented on this second version was real mirrors on the subject's car. To get a render of the environment behind the car, it was necessary to



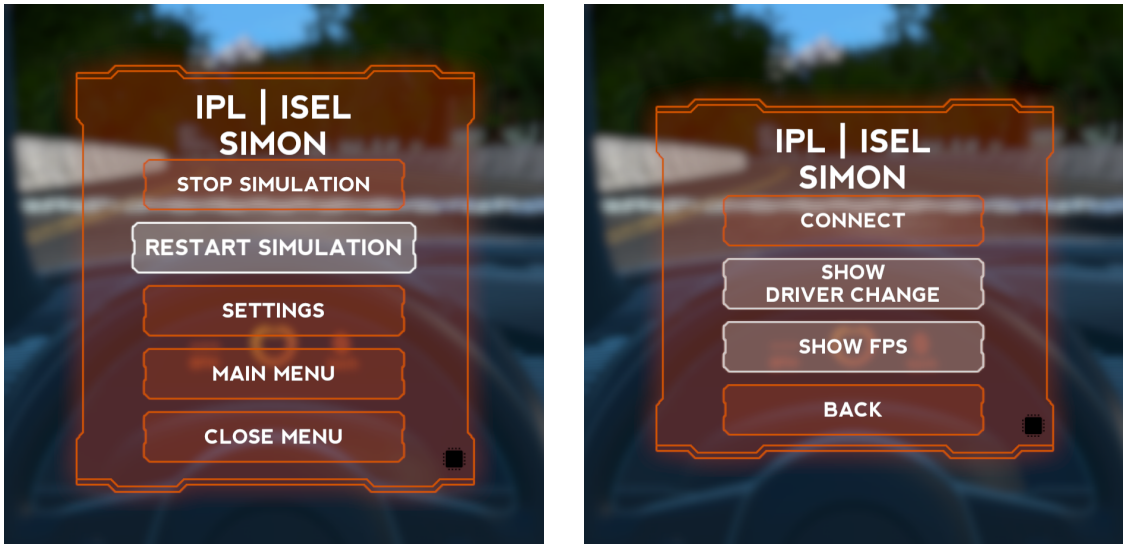
(a) "Welcome New Driver" notification (b) "Fatigue Detected" notification

Figure 5.29: Car front panel notifications

add another camera whose rendering is recorded on a texture that is then mapped on the mirror of the car model.

As expected, this approach lead to a severe loss of performance on the simulator and so, assuming that a real-time image was not essential for a good driving performance, the number of rendered frames per seconds was reduced on the rear camera as an attempt to reduce this effects.

Similarly to the main menu screen, also the simulation environment scene has a menu to control the simulator, with three additional buttons on the main page: *STOP SIMULATION*, *RESTART SIMULATION* and *MAIN MENU*, illustrated by figure 5.30. As expected, the second and third button loads the main menu scene and the simulation environment scene, respectively.



(a) Main page (b) Settings page

Figure 5.30: On simulator screen

On the other hand, the *STOP SIMULATION* button sends a publication in the

`/StoppingSimulation` subject and shows a new menu page that shows an average value of the BPM acquired during the simulation and two images of heart rate signatures, one from the current driver and the other from the previous one, as illustrated on figure 5.31. These images are rendered by the *CardioWheel* module that, in response to the `/StoppingSimulation` publication, saves the result image in a preconfigured directory on the file system and them publishes on the `TemplateReady` subject. Once the simulator receives that publication, it goes to the specified directory and loads the image into a texture object.

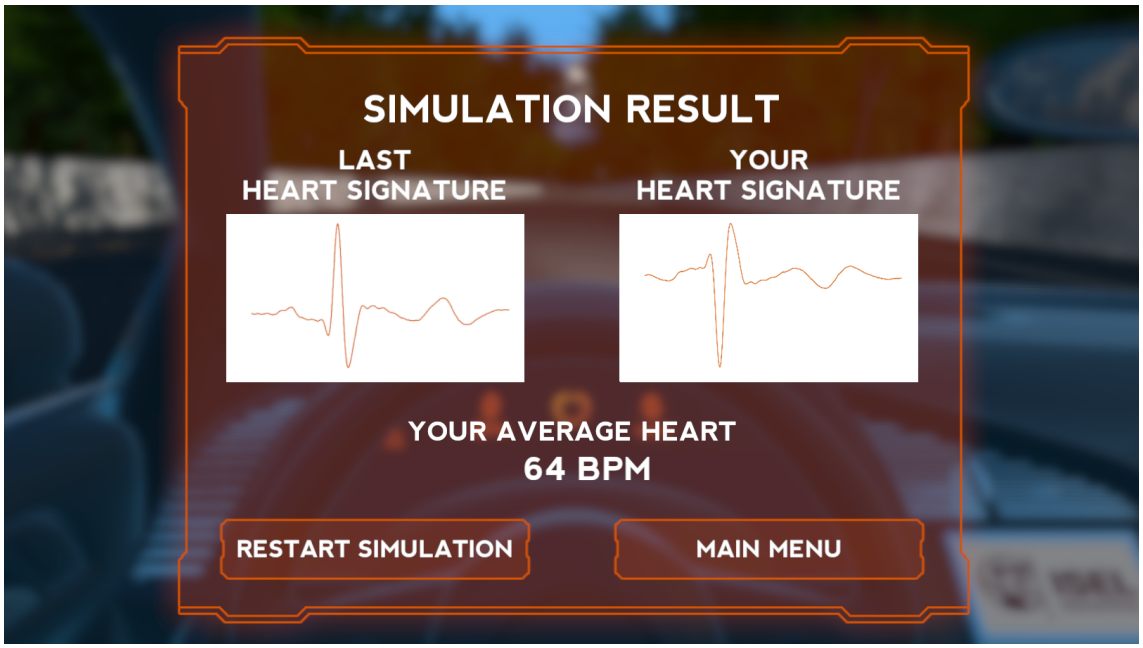


Figure 5.31: Simulation report menu page

6

Experimental Evaluation

As mentioned on chapter 3, the modular system developed would be tested as a tool to acquired data that could successfully be used to classify driving while fatigued or under the influence of alcohol.

Before acquiring any simulator-related data, the communication system must be tested to ensure that it can support the task that it was developed for. On section 5.4 it was mentioned that the simulator has a *dataThoughput* frequency defined at which its information is published, with the exception of sporadic events. As also mentioned, this value directly influences the accuracy of the data sent. For this reason, it was found that at a frequency of 10Hz, the simulator is very lightweight on the communication system and provides good data accuracy.

The *CardioWheel* data can also be considered sporadic since it is not periodic and it is not usually published more than once within a second.

6.1 Communication Architecture Performance

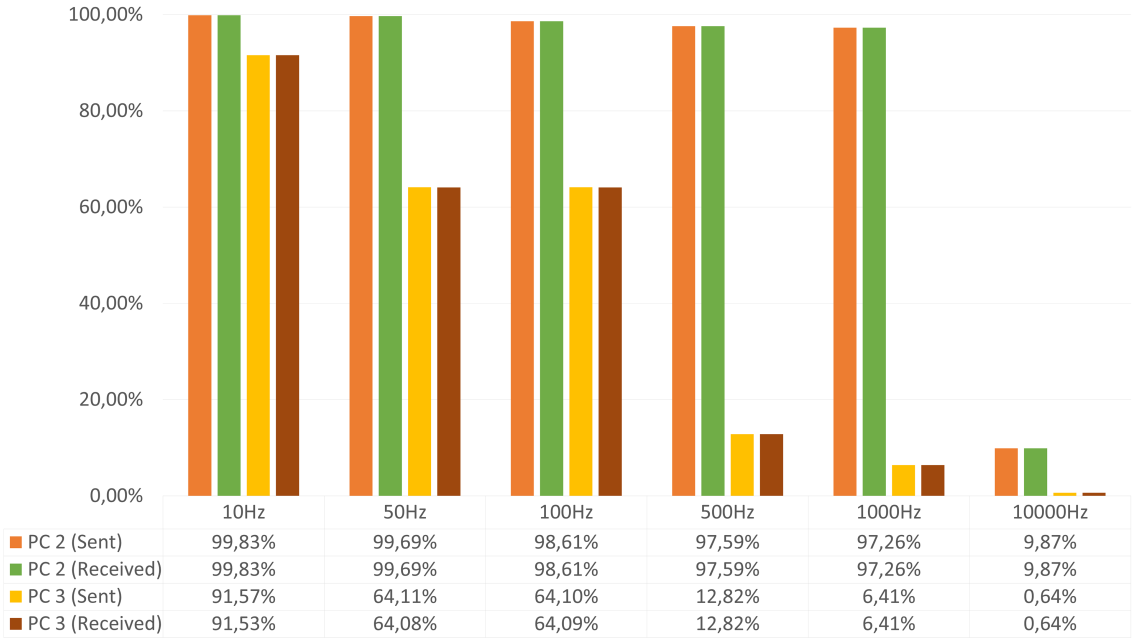
It is necessary to have a relatively good perception of the communication architecture capabilities and for this reason, the platform was tested with three different machines (specifications on table 6.1).

The first set of tests consisted of sending 100-byte packets for a duration of 5 minutes (300 seconds) at a rate of 10Hz, 50Hz, 100Hz, 500Hz, 1kHz and 10kHz,

Table 6.1: Test machine specifications

PC 1 (server)	CPU	Intel Core i7-4770K at 3.5GHz
	RAM	20 GB
	LAN	1Gbps Ethernet connection
	SO	Windows 7 Ultimate 32bit
PC 2	CPU	Intel Core i7-3610QM at 2.3GHz
	RAM	12 GB
	LAN	802.11n Wireless connection at ≈1.5 meter distance
	SO	Windows 10 Pro 64bit
PC 3	CPU	Intel Core 2 Duo E8500 at 3.16GHz
	RAM	3 GB
	LAN	100Mbps Ethernet connection
	SO	Windows 10 Pro 64bit

using PC 1 as the server and PC 2 and PC 3 as clients. The defined packet size was 100 bytes because, in average, that is the size of a message sent and received by the simulator.



	10Hz	50Hz	100Hz	500Hz	1000Hz	10000Hz
PC 2 (Sent)	99,83%	99,69%	98,61%	97,59%	97,26%	9,87%
PC 2 (Received)	99,83%	99,69%	98,61%	97,59%	97,26%	9,87%
PC 3 (Sent)	91,57%	64,11%	64,10%	12,82%	6,41%	0,64%
PC 3 (Received)	91,53%	64,08%	64,09%	12,82%	6,41%	0,64%

Figure 6.1: Communication performance by machine and frequency

It was only detected a negligible packet loss on PC 2, probably due to its Wi-Fi connection. Regarding the ability to send the information at the specified frequency, PC 3 has not able to do it at frequencies above 100Hz. The graph on figure 6.1 illustrates the percentage of information sent and received on each machine, in relation to the theoretical values (table 6.2). Although the drop in performance is noticeable at 50Hz, at 100Hz the machine is only able to send 19'231

of 30'000 packets (64.1%). This bottleneck is obviously repeated at superior frequencies by always sending only 19'231 packets.

Table 6.2: Number of packets expected to be sent

Packet Size	*Sending Rate*	*Number of Packets Expected*
100 bytes	10 Hz	3000
	50 Hz	15000
	100 Hz	30000
	500 Hz	150000
	1000 Hz	300000
	10000 Hz	3000000

As shown on figure 6.1, `PC 2` was also limited to send data above 1kHz. This limitation might be related to timing accuracy and scheduling tasks of the operating system where the tests were executed, since the execution time of the test send operation has timed as ~9.5 microseconds (~105kHz).

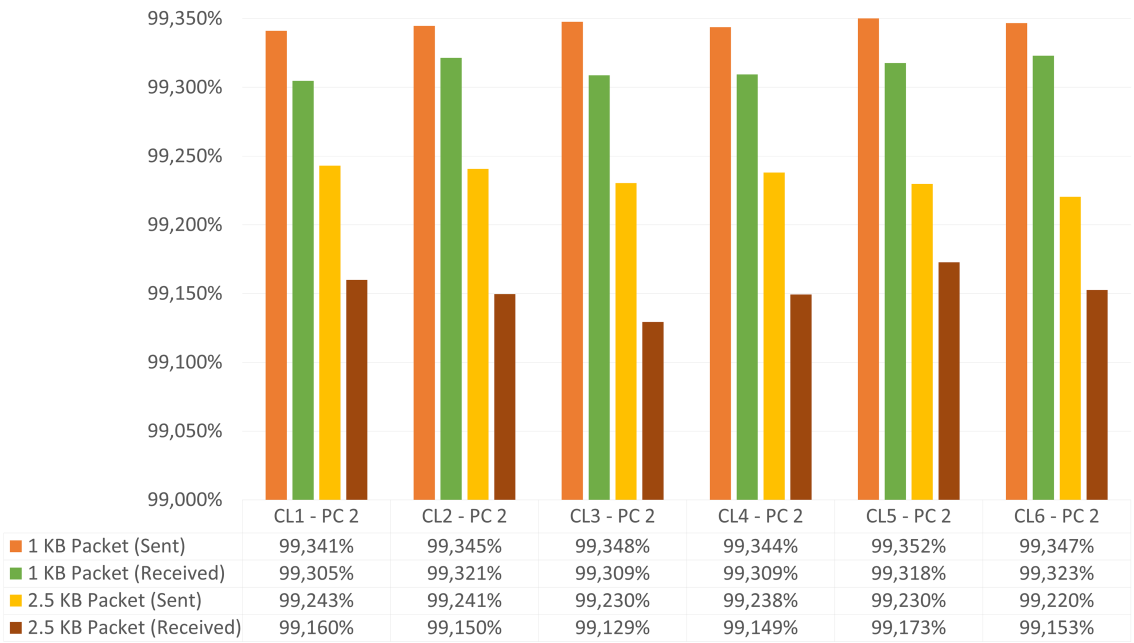


Figure 6.2: Concurrent communication performance by data size

The server, executing on `PC 1` as also tested against six clients from `PC 2`. This test was later prove to be ineffective to test concurrency due to the serialization of the messages on the network interface of the client. Even if it was used multiple machines to host the clients, the server only had one network interface, which means that the messages received would also be serialized by it. To test the concurrency capacity of the server module, it would have to support multiple network interfaces and the information would have to come from clients hosted by

different machines.

Knowing that frequency limit of 1kHz, each client sent 1kB and 2.5kB packets at 1kHz during 5-minute periods. As shown on figure 6.2, the received packet percentage never went below 99.1% (0.9% packet loss) and the server CPU usage was never above 5%. At the end of these tests it was concluded that the server is efficient enough to distribute a lot of information among a number of clients that was reasonable for the purpose of the architecture. However, as expected, its capacity to distribute data is limited by the network bandwidth available.

6.2 Simulator Results

The driving simulation were performed using the setup shown in figure 6.3, which includes a Logitech G29 steering wheel lined with driver's seat. This steering wheel was chosen because it includes a 3-pedal module and it has the ability to rotate 900º from side to side, although this function has been limited to 240º in order to avoid getting your hands off the steering wheel.

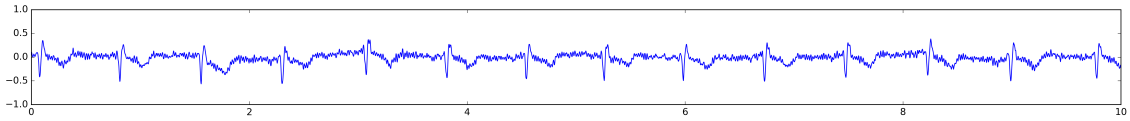
Although there is no defined number of data points used to calculate the measures descriptors, the simulations performed were always at a frame rate of 150Hz or above. This means that, for a send rate of ten messages a second (one message every 100 milliseconds), the minimal amount of data used to calculate the descriptors was fifteen points.

The simulations measured two types of data: physiological and driving performance. As illustrated on figure 6.4(b), some of the data acquired from the ECG signal (physiological data) had too much noise and was not possible to extract features from it. As mentioned during the design process of the simulation track, on section 5.2, this noise could be cause by holding the steering wheel too tighten, by rubbing the hands on it or by moving it too fast or aggressive. Although the ECG examples of figures 6.4(a) and 6.4(b) are not filtered, the amount of noise in the noisy version of the ECG increases the filtering process complexity, making it very difficult to achieve an acceptable result.

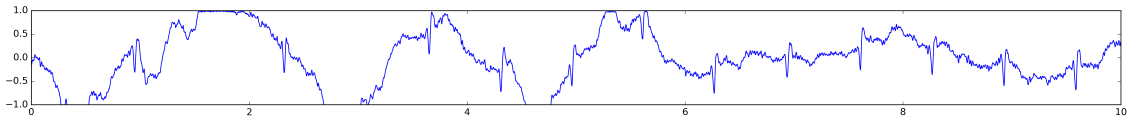
6.2.1 Driving Performance Measurements

Initially, using the first version of the simulator, all the experiments were divided into four laps with a progressive increase of the allowed maximum speed: first lap at low speed ($<$ 70km/h), second lap at mid speed ($<$ 120km/h), third lap

Figure 6.3: Simulator Setup



(a) Non-filtered "Clean" ECG Signal



(b) Non-filtered "Noisy" ECG Signal

Figure 6.4: Non-filtered ECG Signals Examples

at high speed ($<$ 160km/h) and the fourth lap at the maximum speed possible without losing control of the car.

In addition to being executed on the first version of the system, the results obtain were not conclusive due to the experience protocol used and to important measures not being included (e.g. vehicle speed). For this same reason, this data was excluded.

This second version of the system was tested with 10 people (2 women and 8 men) with a mean age of 27 years (standard deviation of 10.5 years). The results

used in this analysis were acquired during 5-minute experiences using the newer version of the system.

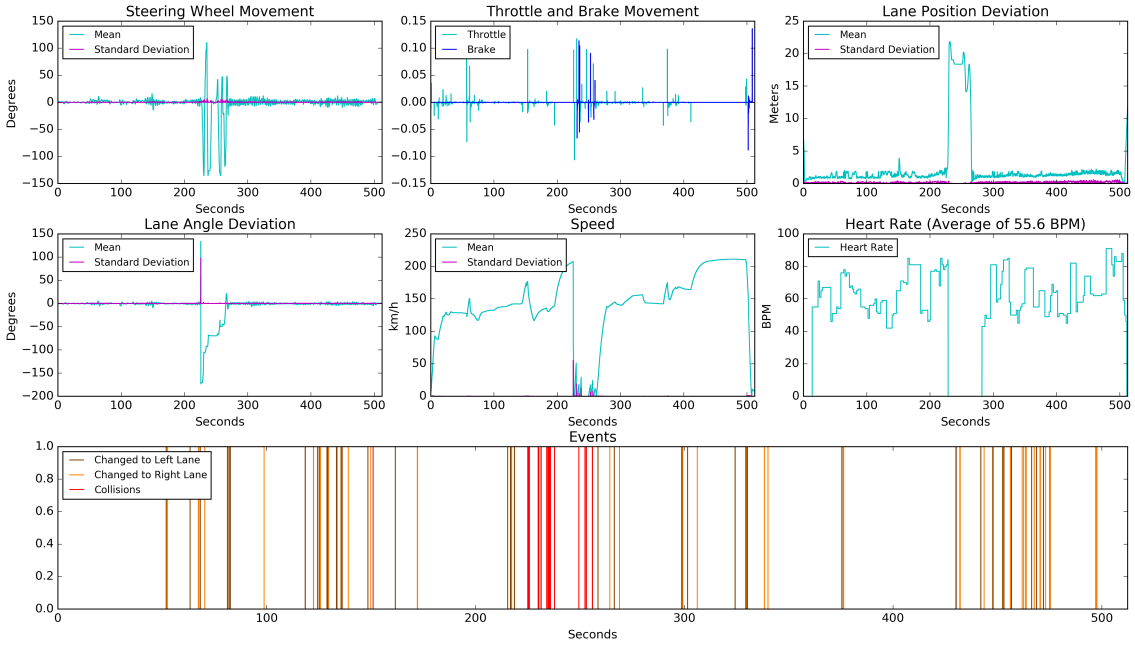


Figure 6.5: *Subject A* - Performance graphs

The information reported by the simulator allows to easily detect the occurrence of collisions, especially when these were of great severity (i.e. too much collision force or a bad collision angle). One good example of it was captured during the experience with *Subject A*, where a severe collision occurred approximately 3 minutes and 45 seconds (225 seconds) after starting the simulation. As illustrated by figure 6.5, all the measures report a abrupt change from approximately 225 to 275 seconds.

Using the *Lane Angle Deviation* it can be concluded that the vehicle change to the opposite direction of the lane. It can also be assumed that the driver released the steering wheel by reading the lack of *Heart Rate* information during that period. The red lines on *Event* graph ensure that this is a truthful conclusion by marking every collision of the vehicle.

Subject B participation was divided into two parts: before and after alcohol consumption of the equivalent to 1.25 litres of beer, which most of the times is not enough to produce a serious effect on an averaged-size person. As illustrated on figures 6.6 and 6.7, the results are very similar, despite the speed difference. This negligence with the speed limit is also noticeable on the *Throttle and Brake Movement* graph, where the low intensity adjustments are almost eliminated.

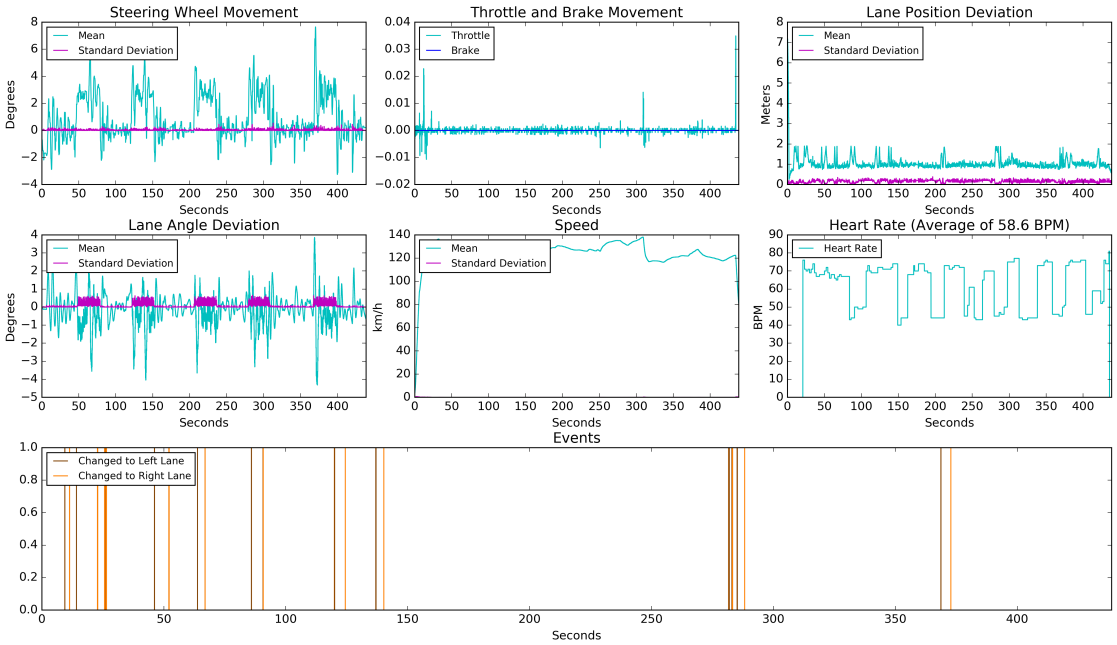


Figure 6.6: *Subject B* - Performance graphs before alcohol consumption

The *Lane Position Deviation* also increases by minimal amount (from ~1.043 to ~1.325 meters), which can be result of alcohol influence.

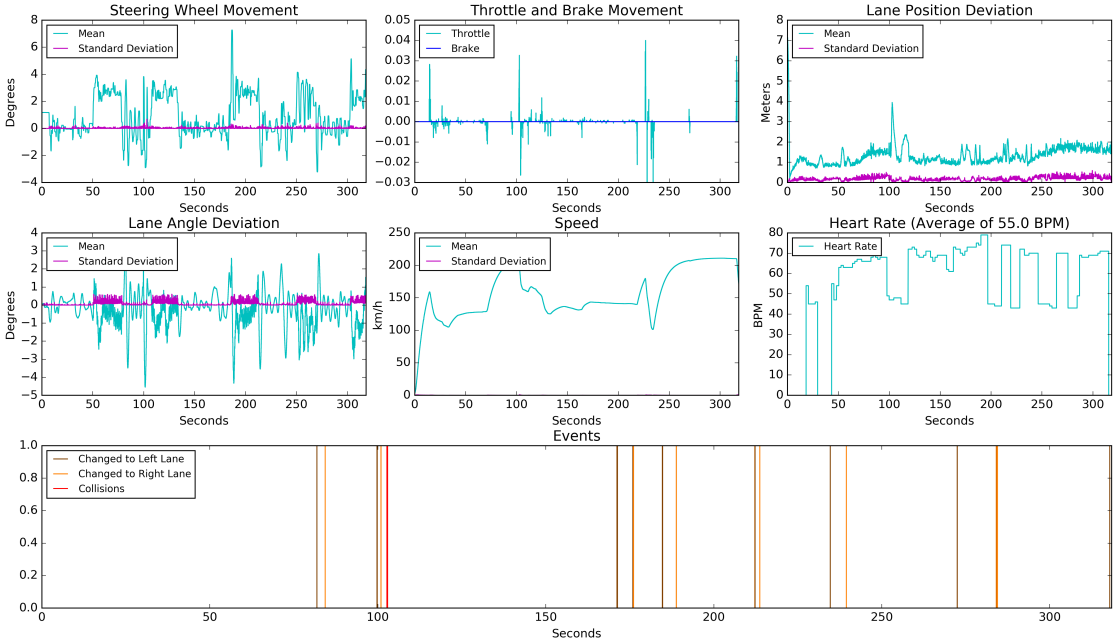


Figure 6.7: *Subject B* - Performance graphs after alcohol consumption

The participation of *Subject C* was very similar to *Subject B* with the addition of a sleep deprivation of 35 hours. On figures 6.8 and 6.9, the graph plots after the consumption of alcohol have the values changing faster than before, which can be assumed to be compensations of exaggerated movements.

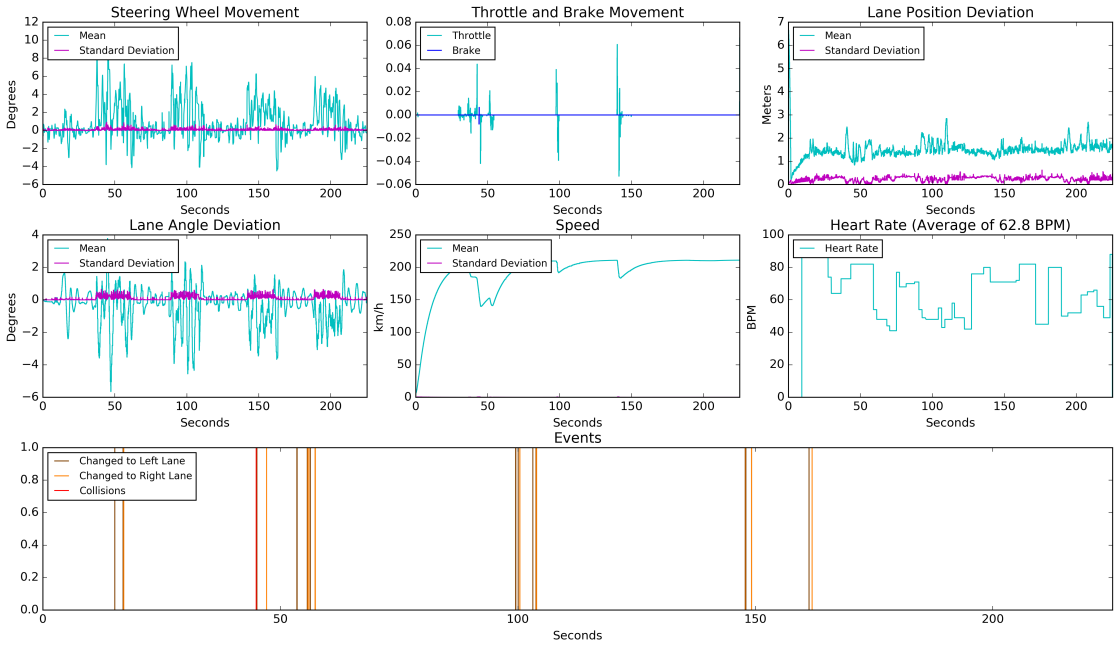


Figure 6.8: *Subject C* - Performance graphs before alcohol consumption

Using the *Event* graphs it is noticeable that, after the consumption of alcohol, the number of collisions increased substantially (from 1 to 25). In addition, most of the other measures also show an small increase, being the most interesting one the *Heart Rate*. Although the mean stayed almost the same, the maximum value of the *Heart Rate* increased by 17 BPM and the values acquired vary more, which is assumed to be related to stress for having a bad performance.

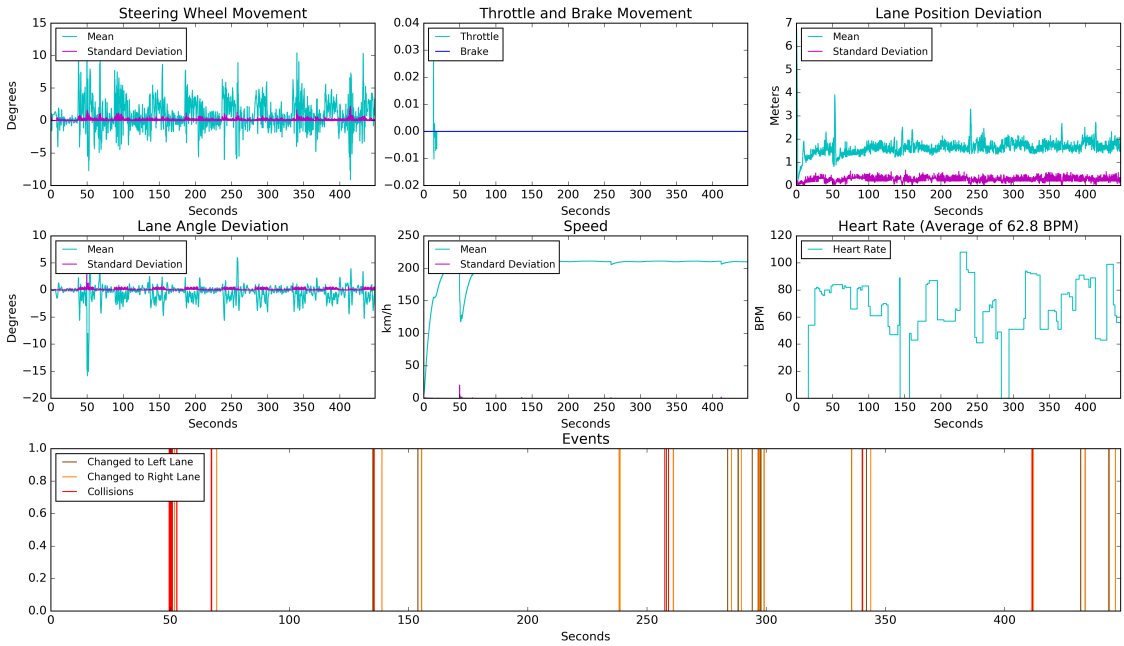


Figure 6.9: *Subject C* - Performance graphs after alcohol consumption

7

Conclusions and Future Work

Although some descriptors plotted on figures 6.5, 6.6, 6.7, 6.8 and 6.9 were not visible, such as the *standard deviation* and the *brake movement*, the analysis of them show that the system has the ability to detect performance changes after the consumption of alcohol. These indications cannot be considered conclusions of the study due to the fact that the test sample is only composed by 10 people.

In order to obtain conclusive results, the test protocol must be must more restrictive and controlled. The tests must also be longer to be possible to fatigue the driver.

The results analysis should also include data classification using machine learning, especially when using data extracted from the ECG signal. It could also be acquired information about the driver's behaviour (e.g. using image processing - section 2.2.2). All this information would contribute to a richer simulation report.

Simulator-wise, the driving component of the system must be revised or completely replaced by other module able to calculate with precision the car and road physics. Some better alternatives than the one included in the framework, accordingly with user reports, would be the packages *Realistic Car Kit* (RCK) [3], *Realistic Car Controller* (RCC) [2], *Edy's Vehicle Physics* (EVP) [1] and *Vehicle Physics Pro* (VPP) [4], which are all paid assets. The last package, in particular, is the best candidate since it would provide a professional and in-depth tweak capability and as been shown to out perform many competitors [60]. Figure 7.1 exemplifies

the tire tuning capability of the VPP package, which are one of the main components for a good driving simulation.

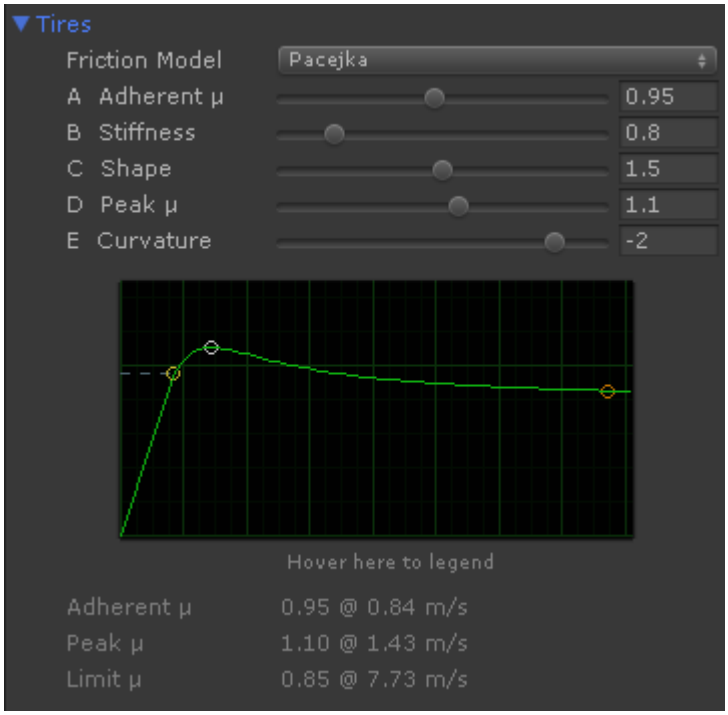


Figure 7.1: Tires tuning capability

Still related to the simulator, the environment also have to be optimized it performs smoothly without the need of a top-tear computer. Graphical and sound redesigns could also give the driver a better involvement in the virtual environment, which would make the experiment much more realistic and reliable.

Finally, the optimal goal track-wise would be to implement a procedural generator to allow the driver to experience a "infinite" non-cyclic road. This goal would be even better with the integration with multi-theme environments, also procedural generated.

The communication architecture developed also have some components that need to be refactored, fixed or even completely redesign in order to make it more robust. On a future version, the *OSCD* server would have a graphical user interface with a session log, with permanent registry of it in a database, and access and management of the subjects registered and subscribed by each user.

Still related to the communication architecture, the *OSCD* protocol develop need to have more commands added to improve its capability of expansion. One useful feature to be implement would be the possibility of a subscription to be in

pull, where the client requests the latest publication, or in push mode, where the server sends automatically the data.

This dissertation has also produced an article with the title "*Driving Simulator for Performance Monitoring with Physiological Sensor*", submitted in November 2017 to the **19th IEEE Mediterranean Electrotechnical Conference (MELECON) 2018**. This article focus in the usage of the simulator as a tool to support the development of advanced driver assisting systems (ADAS), which are helping to reduce the number of accidents and to improve safety on roads.

As mentioned on section 1.2, this project is open to be used and extended. The following URL can be used to access the code repository produced during the development of this thesis. Due to copyright restrictions of some components used in the simulator project, only the *OSCD* system is currently available.

`https://bitbucket.org/account/user/simon_team/projects/SIMON`

Bibliography

[1] Unity asset store: Edy's vehicle physics, 2011. URL https://www.assetstore.unity3d.com/en/content/403. (p. 93)

[2] Unity asset store: Realistic car controller, 2014. URL https://www.assetstore.unity3d.com/en/content/16296. (p. 93)

[3] Unity asset store: Realistic car kit, 2014. URL https://www.assetstore.unity3d.com/en/content/18421. (p. 93)

[4] Vehicle physics pro official blog, 2014. URL http://vehiclephysics.com/. (p. 93)

[5] Unity asset store: Standard assets, 2015. URL https://www.assetstore.unity3d.com/en/content/32351. (p. 57)

[6] Official website of *SCS Software*, 2017. URL http://scssoft.com. (p. 23)

[7] Official website of *Unity*, 2017. URL https://unity3d.com. (p. 26)

[8] Página wikipedia da *Warthog Games*, 2017. URL https://en.wikipedia.org/wiki/Warthog_Games. (p. 21)

[9] National Highway Traffic Safety Administration et al. Drowsy driving and automobile crashes. *NCSDR/NHTSA Expert Panel on Driver Fatigue and Sleepiness. DOT Report HS*, 808:707, 1998. (p. 3)

[10] Mehmet Akin, Muhammed B. Kurt, Necmettin Sezgin, and Muhittin Bayram. Estimating vigilance level by using eeg and emg signals. *Neural Computing and Applications*, 17(3):227–236, 2008. (pp. 6, 17 e 19)

[11] Fabio Baladez. O passado, o presente e o futuro dos simuladores. *FaSCi-Tech*, 1(1), 2016. (p. 1)

[12] Kevin C. Baldwin, Donald D. Duncan, and Sheila K. West. The driver monitor system: A means of assessing driver performance. *Johns Hopkins APL Technical Digest*, 25(3):269–277, 2004. (pp. 2 e 4)

[13] João B. P. Begonha. Auto-estradas - características técnicas, 2008. (p. 62)

[14] I.-D. Brown, A.-H. Tickner, and D.-C.-V. Simmonds. Effect of prolonged driving on overtaking criteria. *Ergonomics*, 13(2):239–242, 1970. (p. 3)

[15] CDC. Impaired driving: Get the facts, 2017. URL https://www.cdc.gov/motorvehiclesafety/impaired_driving/impaired-drv_factsheet.html. (p. 4)

[16] John Cohen, E. J. Dearnaley, and C. E. M. Hansel. The risk taken in driving under the influence of alcohol. *Br Med J*, 1(5085):1438–1442, 1958. ISSN 0007-1447. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2029328/. 13536526[pmid]. (p. 4)

[17] Drew Dawson and Kathryn Reid. Fatigue, alcohol and performance impairment. *Nature*, 388(6639):235, 1997. (pp. 4 e 26)

[18] Helios de Rosario Martínez, José S. Solaz Sanahuja, Elisa Signes i Pérez, Andrés Soler Valero, Nicolás Palomares Olivares, Jordi Jornet, Marc Dominguis, Jorge Juan Gil Nobajas, Iñaki Díaz Garmendia, and Mikel Echeverría Larrañaga. Haptic technologies for improving driving safety, 2011. URL http://gestion.ibv.org/gestoribv/index.php?option=com_docmanview=download&alias=359-haptic-technologies-for-improving-driving-safety&category_slug=productos&Itemid=142. (p. 12)

[19] V.-V. Dementienko, V.-B. Dorokhov, L.-G. Koreneva, A.-G. Markov, and V.-M. Shakhnarovich. The characteristics of the electrodermal activity during changes in the level of human wakefulness. *Zhurnal vysshei nervnoi deiatelnosti imeni I P Pavlova*, 49(6):926–935, 1999. ISSN 0044-4677. URL http://europepmc.org/abstract/MED/10693272. (p. 17)

[20] D. Djaouti, J. Alvarez, J.-P. Jessel, and O. Rampnoux. Origins of serious games, 2008. URL http://www.ludoscience.com/files/ressources/origins_of_serious_games.pdf. (p. 26)

[21] City Car Driving. Official website of *City Car Driving*, 2017. URL http://citycardriving.com. (p. 22)

[22] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Http - status code definitions. Technical report, 1999. URL https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html. (p. 36)

[23] Theodore W. Forbes. The influence of driver fatigue on traffic accidents. *Driver fatigue in road traffic accidents*, page 33, 1978. (p. 3)

[24] Blender Foundation. Official website of *Blender*, 2017. URL https://www.blender.org. (p. 60)

[25] GameDesigning. Blog gamedesigning, 2017. URL https://www.gamedesigning.org. (p. 26)

[26] Kathryn Graham. Theories of intoxicated aggression. *Canadian Journal of Behavioural Science/Revue canadienne des sciences du comportement*, 12(2):141, 1980. (p. 4)

[27] Takchito Hayami, Katsuya Matsunaga, Kazunori Shidoji, and Yuji Matsuki. Detecting drowsiness while driving by measuring eye movement-a pilot study. In *The IEEE 5th International Conference on Intelligent Transportation Systems*, pages 156–161. IEEE, 2002. (p. 15)

[28] Jennifer A. Healey and Rosalind W. Picard. Detecting stress during real-world driving tasks using physiological sensors. *IEEE Transactions on intelligent transportation systems*, 6(2):156–166, 2005. (p. 13)

[29] Michael G. Hinchey, Roy Sterritt, and Chris Rouff. Swarms and swarm intelligence. *Computer*, 40(4), 2007. (p. 67)

[30] E. Hoddes, V. Zarcone, H. Smythe, R. Phillips, and W. C. Dement. Quantification of sleepiness: a new approach. *Psychophysiology*, 10(4):431–436, 1973. (p. 13)

[31] Liberty Hoekstra-Atwood. Driving under involuntary and voluntary distraction: Individual differences and effects on driving performance, 2015. URL https://hfast.mie.utoronto.ca/wp-content/uploads/Publications/Hoekstra-Atwood_Liberty_201511_MASc_thesis.pdf. (p. 12)

[32] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321200683. (p. 28)

[33] Patricia K. Immich, Ravi S. Bhagavatula, and Ravi Pendse. Performance analysis of five interprocess communication mechanisms across unix operating systems. *Journal of Systems and Software*, 68(1):27–43, 2003. (p. 27)

[34] iMotions. Eye tracking & facial expression analysis in a driving simulator, 2014. URL https://imotions.com/blog/eye-tracking-facial-expression-analysis-driving-simulator/. (p. 11)

[35] Image Space Incorporated. Official website of *Image Space Incorporated*, 2017. URL http://imagespaceinc.com. (p. 20)

[36] Michael Ingre, Torbjörn Åkerstedt, Björn Peters, Anna Anund, and Göran Kecklund. Subjective sleepiness, simulated driving performance and blink duration: examining individual differences. *Journal of sleep research*, 15(1): 47–53, 2006. (p. 14)

[37] iRacing. Official website of *iRacing*, 2017. URL http://www.iracing.com. (p. 21)

[38] Bashkim Isai. Message queues: Comparing beanstalkd, ironmq and amazon sqs, 2014. URL https://www.sitepoint.com/message-queues-comparing-beanstalkd-ironmq-amazon-sqs/. (p. 27)

[39] Kosuke Kaida, Masaya Takahashi, Torbjörn Åkerstedt, Akinori Nakata, Yasumasa Otsuka, Takashi Haratani, and Kenji Fukasawa. Validation of the karolinska sleepiness scale against performance and eeg variables. *Clinical Neurophysiology*, 117(7):1574–1581, 2006. (p. 14)

[40] Nico Kaptein, Jan Theeuwes, and Richard Van Der Horst. Driving simulator validity: Some considerations. *Transportation Research Record: Journal of the Transportation Research Board*, (1550):30–36, 1996. (p. 10)

[41] Rami N. Khushaba, Sarath Kodagoda, Sara Lal, and Gamini Dissanayake. Driver drowsiness classification using fuzzy wavelet-packet-based feature-extraction algorithm. *IEEE Transactions on Biomedical Engineering*, 58(1):121–131, 2011. (p. 17)

[42] A. K. Kokonozi, E. M. Michail, I. C. Chouvarda, and N. M. Maglaveras. A study of heart rate and brain system complexity and their interaction in sleep-deprived subjects. In *Computers in Cardiology, 2008*, pages 969–971. IEEE, 2008. (pp. 6 e 17)

[43] Hans Laurell and H.-O. Lisper. Changes in subsidiary reaction time and heart-rate during car driving, passenger travel and stationary conditions. *Ergonomics*, 19(2):149–156, 1976. (p. 3)

[44] B. G. Lee, S. J. Jung, and W. Y. Chung. Real-time physiological and vision monitoring of vehicle driver for non-intrusive drowsiness detection. *IET Communications*, 5:2461–2469, 2011. ISSN 1751-8628. URL http://digital-library.theiet.org/content/journals/10.1049/iet-com.2010.0925. (p. 17)

[45] Gang Li and Wan-Young Chung. Detection of driver drowsiness using wavelet analysis of heart rate variability and a support vector machine classifier. *Sensors*, 13(12):16494–16511, 2013. ISSN 1424-8220. doi: 10.3390/s131216494. URL http://www.mdpi.com/1424-8220/13/12/16494. (p. 17)

[46] Wen Chieh Liang, John Yuan, Deh Chuan Sun, and Ming Han Lin. Changes in physiological parameters induced by indoor simulated driving: Effect of lower body exercise at mid-term break. *Sensors*, 9(9):6913–6933, 2009. ISSN 1424-8220. doi: 10.3390/s90906913. URL http://www.mdpi.com/1424-8220/9/9/6913. (p. 18)

[47] Eduardo G. Lima. Álcool, estupefacientes e sinistralidade rodoviária. *CEDIS Working Papers*, (23), 2015. (p. 4)

[48] Charles C. Liu, Simon G. Hosking, and Michael G. Lenné. Predicting driver drowsiness using vehicle measures: Recent insights and future challenges. *Journal of safety research*, 40(4):239–245, 2009. (p. 16)

[49] Silvano Maffeis and Douglas C. Schmidt. Constructing reliable distributed communication systems with corba. *IEEE Communications Magazine*, 35(2): 56–60, 1997. (p. 27)

[50] Microsoft. Integration topologies: Message broker pattern, 2004. URL https://msdn.microsoft.com/en-us/library/ff648849.aspx. (p. 29)

[51] Microsoft. Integration topologies: Publish/subscribe pattern, 2004. URL https://msdn.microsoft.com/en-us/library/ff649664.aspx. (p. 28)

[52] MobilEye. Official website of *MobilEye*, 2017. URL http://www.mobileye.com/. (p. 16)

[53] Compumedics Neuroscan. Official website of *Compumedics Neuroscan*, 2017. URL http://compumedicsneuroscan.com/. (p. 12)

[54] James. O'Malley. 5 racing games that nailed realistic driving physics – and 3 that didn't. *TechRadar*, 2015. URL http://www.techradar.com/news/car-tech/5-racing-games-that-nailed-realistic-driving-physics-and-3-that-didn-t-1305257. (p. 20)

[55] M. Patel, S.-K.-L. Lal, Diarmuid Kavanagh, and Peter Rossiter. Applying neural network analysis on heart rate variability data to assess driver fatigue. *Expert systems with Applications*, 38(6):7235–7242, 2011. (pp. 17 e 18)

[56] P. Philip, P. Sagaspe, J. Taillard, C. Valtat, N. Moore, T. Åkerstedt, A. Charles, and B. Bioulac. Fatigue, sleepiness, and performance in simulated versus real driving conditions. *SLEEP*, 28(12), 2005. URL http://www.journalsleep.org/Articles/281206.pdf. (p. 3)

[57] Pierre Philip, F. Vervialle, P. Le Breton, Jacques Taillard, and James A. Horne. Fatigue, alcohol, and serious road crashes in france: factorial study of national data. *Bmj*, 322(7290):829–830, 2001. (p. 26)

[58] Philip M. Podsakoff and Dennis W. Organ. Self-reports in organizational research: Problems and prospects. *Journal of Management*, 12(4):531–544, 1986. doi: 10.1177/014920638601200408. URL https://doi.org/10.1177/014920638601200408. (pp. 2 e 13)

[59] Tapan Pradhan, Ashutosh Nandan Bagaria, and Aurobinda Routray. Measurement of perclos using eigen-eyes. In *4th International Conference on Intelligent Human Computer Interaction (IHCI)*, pages 1–4. IEEE, 2012. (p. 15)

[60] Vehicle Physics Pro. Comparing vpp with other vehicle kits, 2017. URL http://vehiclephysics.com/about/comparison/. (p. 93)

[61] Lets Race. Official website of *Lets Race*, 2017. URL http://www.letsrace.co.uk. (p. 20)

[62] Craig W. Reynolds. Steering behaviors for autonomous characters. In *Game developers conference*, volume 1999, pages 763–782, 1999. (pp. xvii, 67 e 68)

[63] Helios De Rosario, José S. Solaz, Noelia Rodríguez, and Luis M. Bergasa. Controlled inducement and measurement of drowsiness in a driving simulator. 2010. URL http://harken.ibv.org/index.php/documents/doc_download/80-controlled-inducement-and-measurement-of-drowsiness-in-a-driving-simulator. (p. 10)

[64] Arun Sahayadhas, Kenneth Sundaraj, and Murugappan Murugappan. Detecting driver drowsiness based on sensors: a review. *Sensors*, 12(12):16937–16953, 2012. (pp. 2, 3, 6, 13 e 17)

[65] Lisa N. Sharwood, Jane Elkington, Mark Stevenson, Ronald R. Grunstein, Lynn Meuleners, Rebecca Q. Ivers, Narelle Haworth, Robyn Norton, and Keith K. Wong. Assessing sleepiness and sleep disorders in australian long-distance commercial vehicle drivers: Self-report versus an "at home" monitoring device. *Sleep*, 35(4):469–475, 2012. doi: 10.5665/sleep.1726. URL http://dx.doi.org/10.5665/sleep.1726. (p. 13)

[66] Daniel Shiffman, Shannon Fry, and Zannah Marsh. *The nature of code*. D. Shiffman, 2012. (p. 67)

[67] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014. (p. 27)

[68] The National Advanced Driving Simulator. Nads-1, 2017. URL http://www.nads-sc.uiowa.edu/sim_nads1.php. (p. 11)

[69] The National Advanced Driving Simulator. Nads-2, 2017. URL http://www.nads-sc.uiowa.edu/sim_nads2.php. (p. 11)

[70] The National Advanced Driving Simulator. Nads minisim, 2017. URL http://www.nads-sc.uiowa.edu/sim_minisim.php. (p. 11)

[71] Kunos Simulazioni. Official website of *Kunos Simulazioni*, 2017. URL http://www.kunos-simulazioni.com. (p. 20)

[72] Santokh Singh. Traffic safety facts - critical reasons for crashes investigated in the national motor vehicle crash causation survey, 2015. URL https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812115. (p. 2)

[73] D. Sommer and M. Golz. Evaluation of perclos based current fatigue monitoring technologies. In *Engineering in Medicine and Biology Society (EMBC), 2010 annual international conference of the IEEE*, pages 4456–4459. IEEE, 2010. (pp. 15 e 16)

[74] Slightly Mad Studios. Official website of *Slightly Mad Studios*, 2017. URL http://www.slightlymadstudios.com. (p. 20)

[75] Juan Sztajzel et al. Heart rate variability: a noninvasive electrocardiographic method to measure the autonomic nervous system. *Swiss medical weekly*, 134 (35-36):514–522, 2004. (p. 18)

[76] Lana M. Trick and James T. Enns. A two-dimensional framework for understanding the role of attentional selection in driving. *Human factors of visual and cognitive performance in driving*, pages 63–73, 2009. (pp. xxi e 12)

[77] Johannes van den Berg, Gregory Neely, Leif Nilsson, Anders Knutsson, and Ulf Landström. Electroencephalography and subjective ratings of sleep deprivation. *Sleep medicine*, 6(3):231–240, 2005. (p. 14)

[78] Matthew Wright. Open sound control: an enabling technology for musical networking. *Organised Sound*, 10(3):193–200, 2005. doi: 10.1017/S1355771805000932. (p. 31)

[79] Jian-Feng Xie, Mei Xie, and Wei Zhu. Driver fatigue detection based on head gesture and perclos. In *International Conference on Wavelet Active Media Technology and Information Processing (ICWAMTIP)*, pages 128–131. IEEE, 2012. (p. 15)

[80] C. Zhao, M. Zhao, J. Liu, and C. Zheng. Electroencephalogram and electrocardiograph assessment of mental fatigue in a driving simulator. *Accident Analysis & Prevention*, 45:83–90, 2011. doi: 10.1016/j.aap.2011.11.019. URL https://www.ncbi.nlm.nih.gov/pubmed/22269488. (p. 11)

CardioWheel Server Base Structure

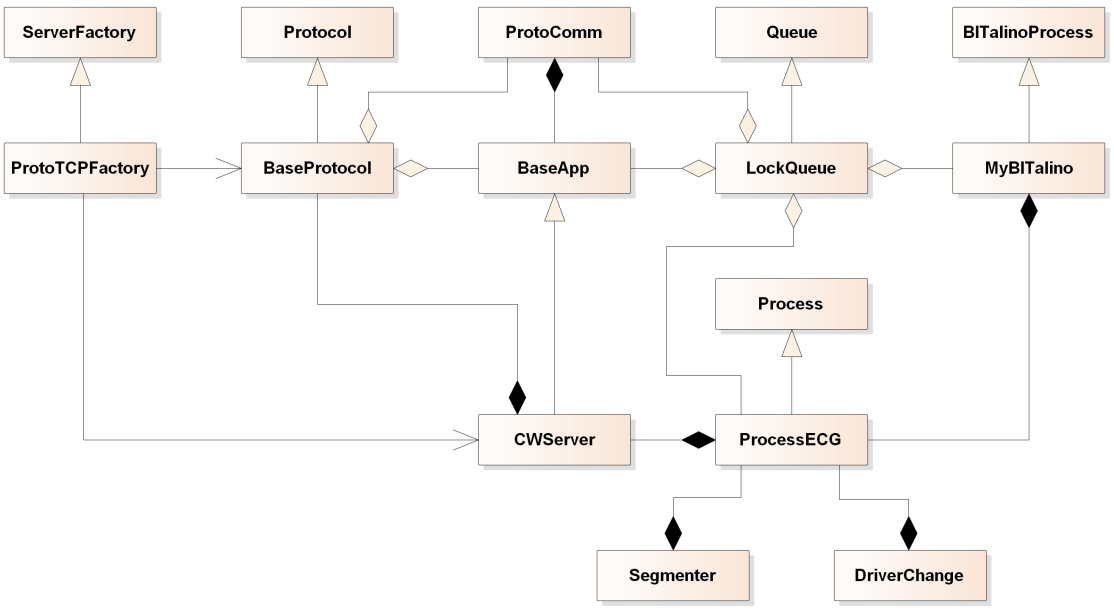
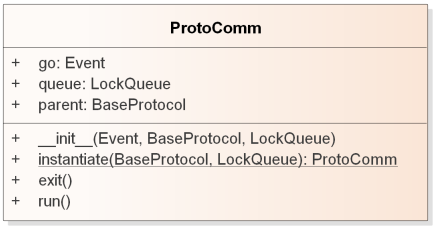
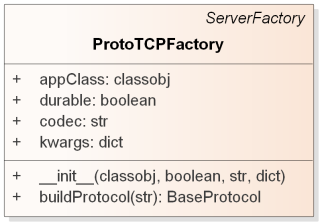


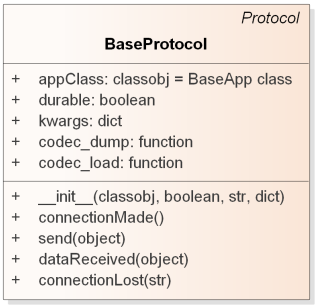
Figure A.1: Structure of the *CardioWheel* Server - Overview



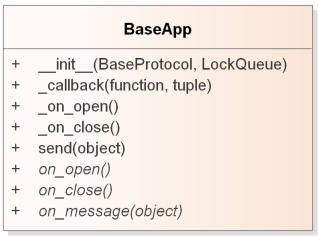
(a) `ProtoComm`



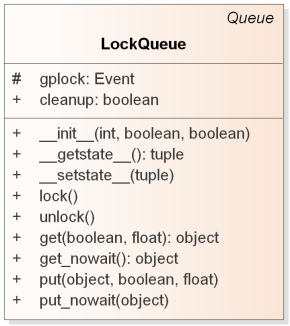
(b) `ProtoTCPFactory`



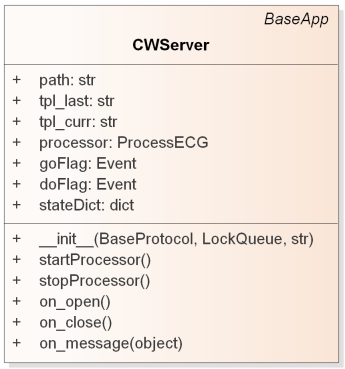
(c) `BaseProtocol`



(d) `BaseApp`



(e) `LockQueue`



(f) `CWServer`

Figure A.2: CardioWheel main classes

B

OSC Distributor Commands Without Server Response

Unregister Subject Command

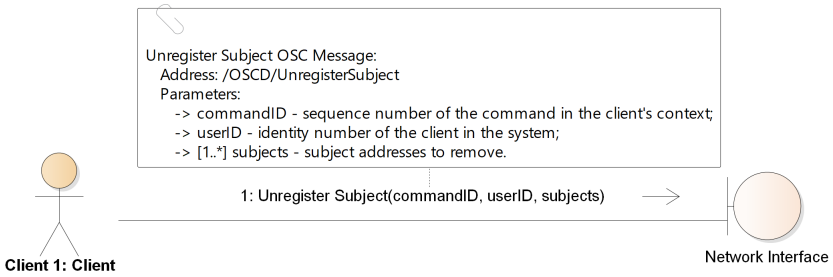


Figure B.1: Communication model of a *Unregister Subject* request

Unsubscribe Subject Command

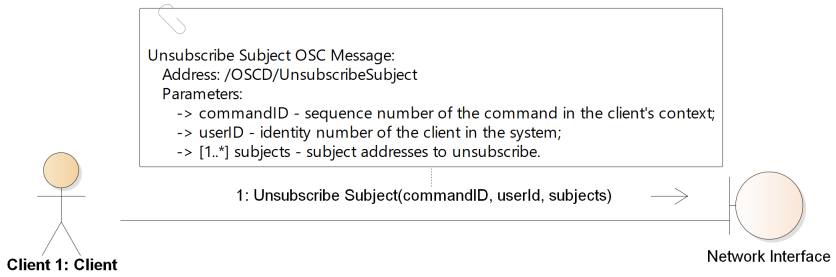


Figure B.2: Communication model of a *Unsubscribe Subject* request

Enable Subject Notification Command

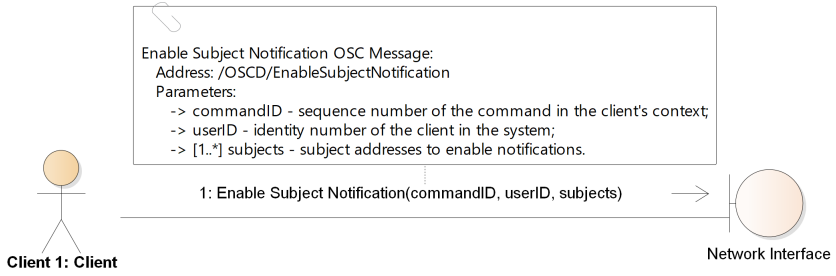


Figure B.3: Communication model of a *Enable Subject Notification* request

Disable Subject Notification Command

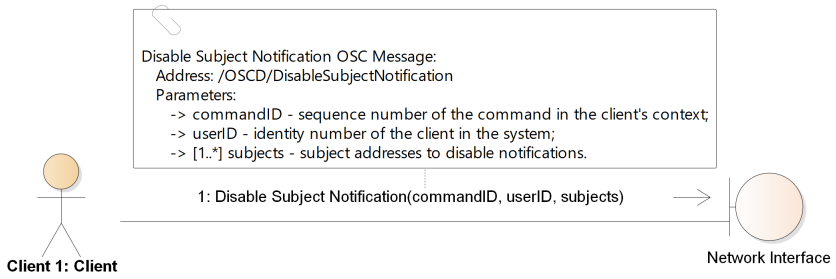


Figure B.4: Communication model of a *Disable Subject Notification* request

C

OSC Distributor Base Structure

`Core` Package

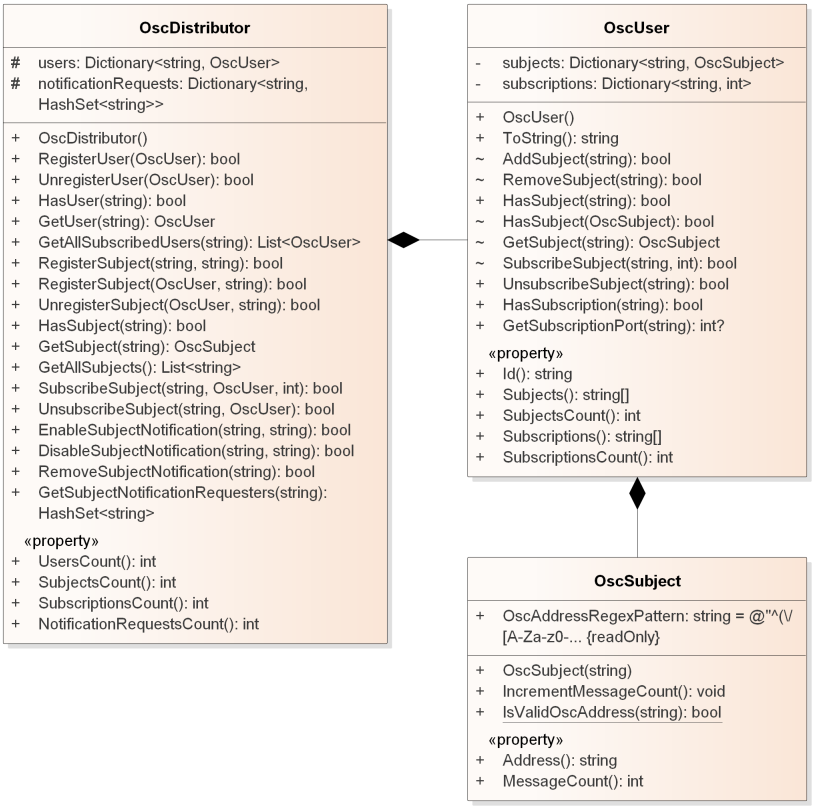


Figure C.1: Structure of the OSC Distributor

`Network` Package

Figure C.2: Structure of the OSC Network Distributor

D

OSCD C# Client Base Structure

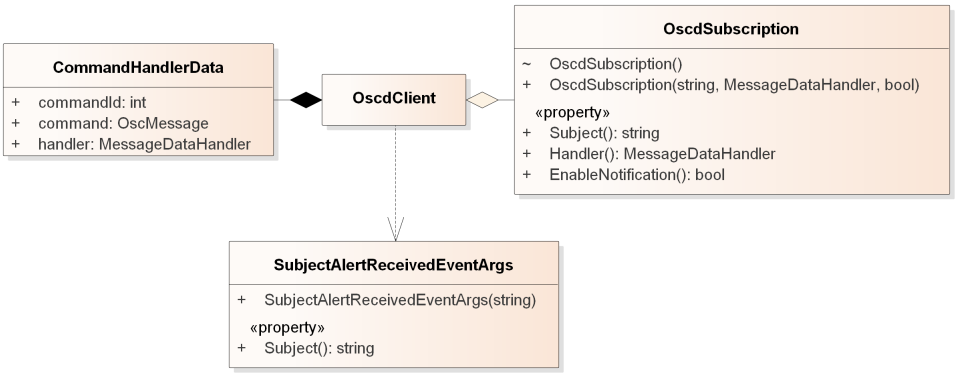


Figure D.1: (Simplified) Structure of the OSCD C# Client

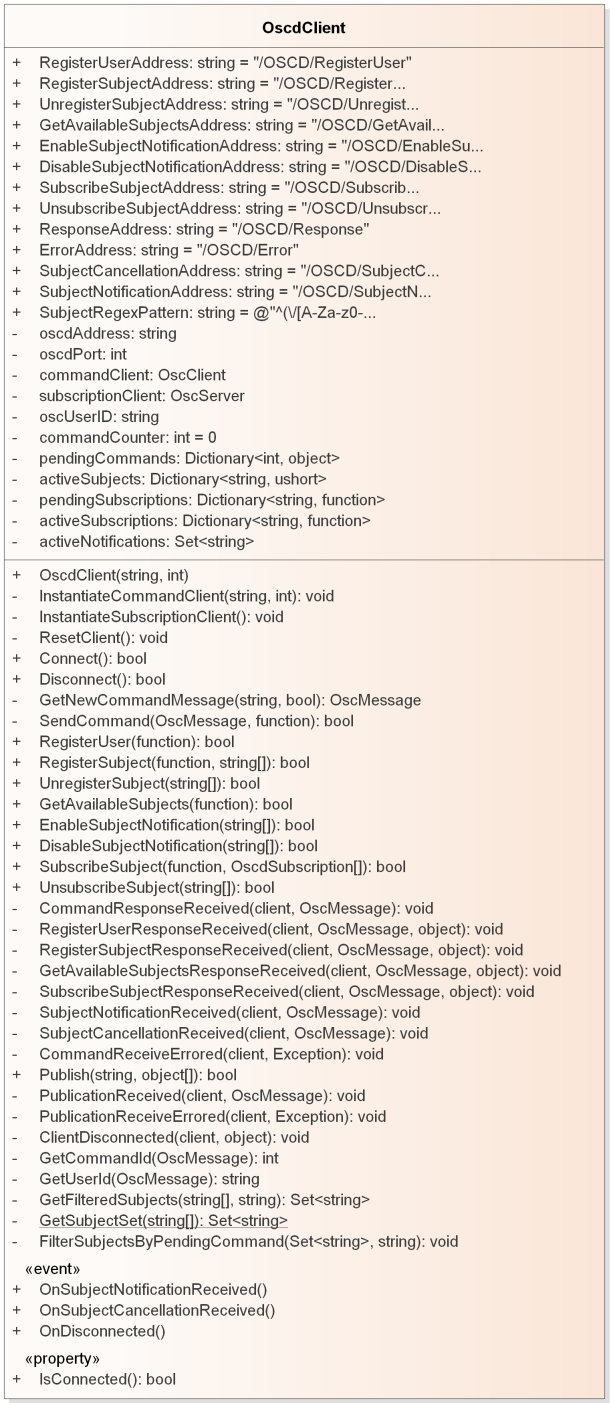


Figure D.2: `OscdClient` class

E

Boids Package

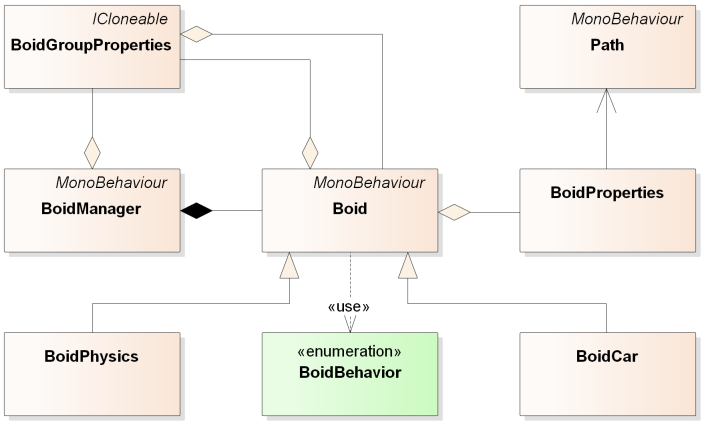


Figure E.1: (Simplified) Structure of the `Boids` package

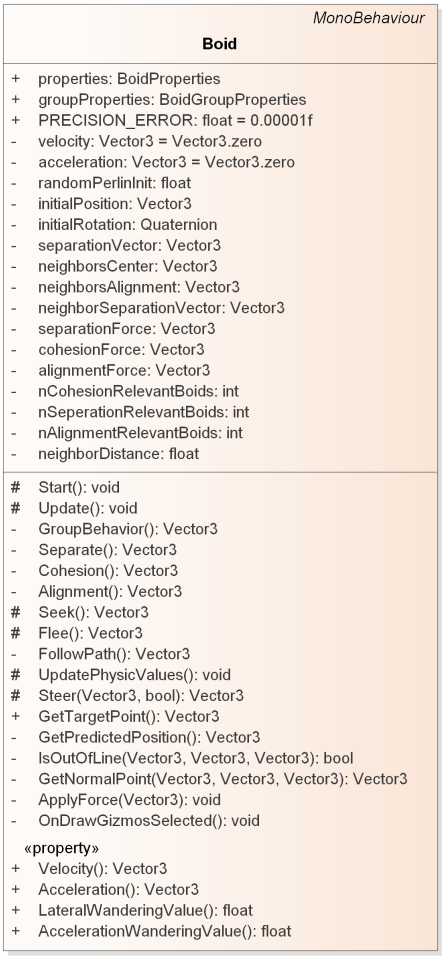
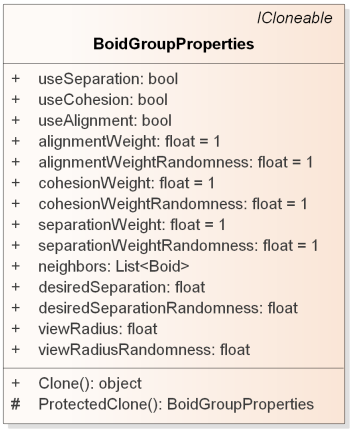
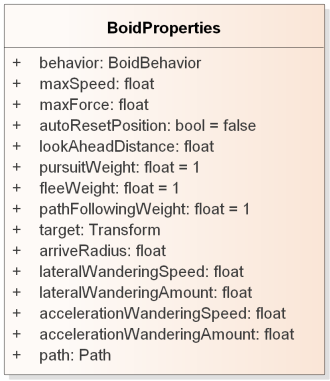


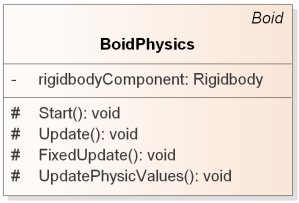
Figure E.2: `Boid` class



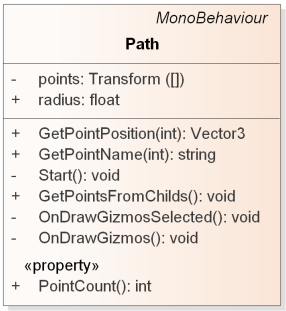
(a) `BoidGroupProperties`



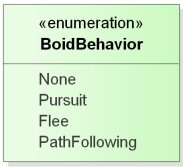
(b) `BoidProperties`



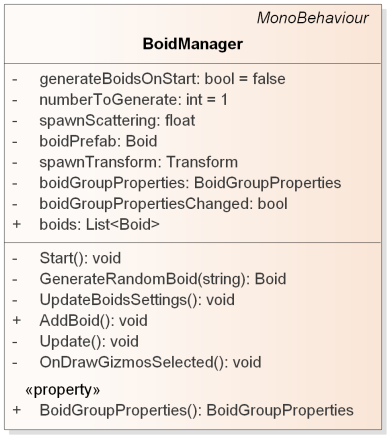
(c) `BoidPhysics`



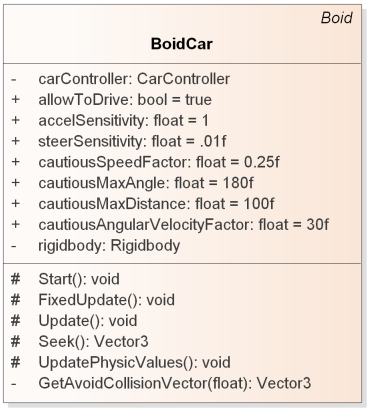
(d) `Path`



(e) `BoidBehavior`



(f) `BoidManager`



(g) `BoidCar`

Figure E.3: `Boid` package classes (without `Boid`)

F

`BITalinoClient` Package

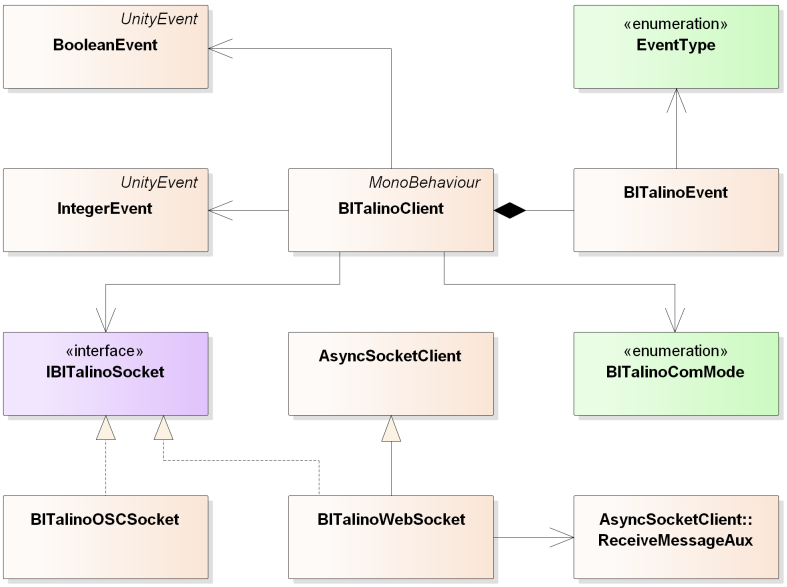


Figure F.1: (Simplified) Structure of the `BITalinoClient` package

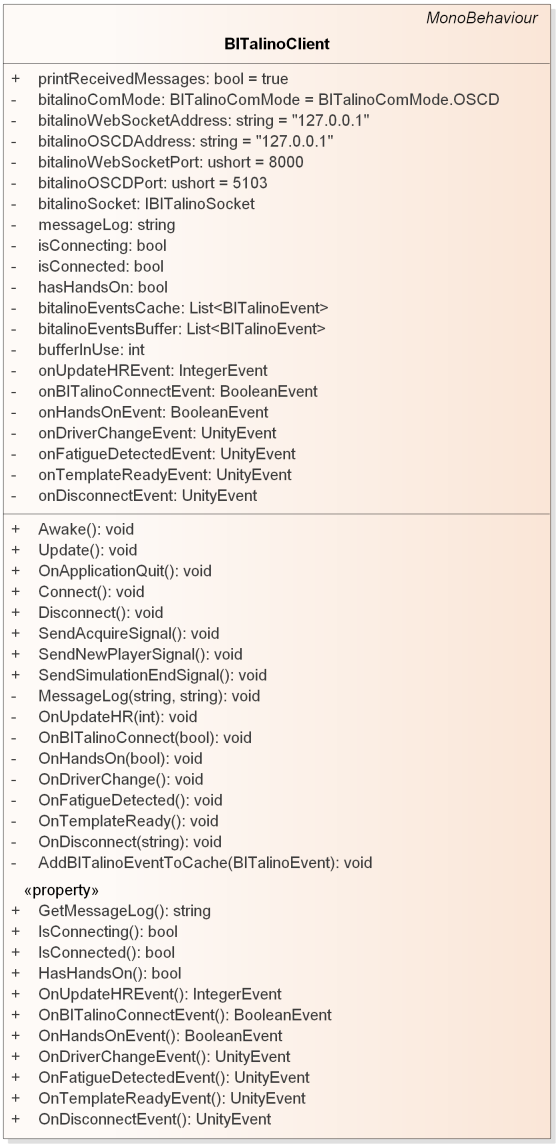
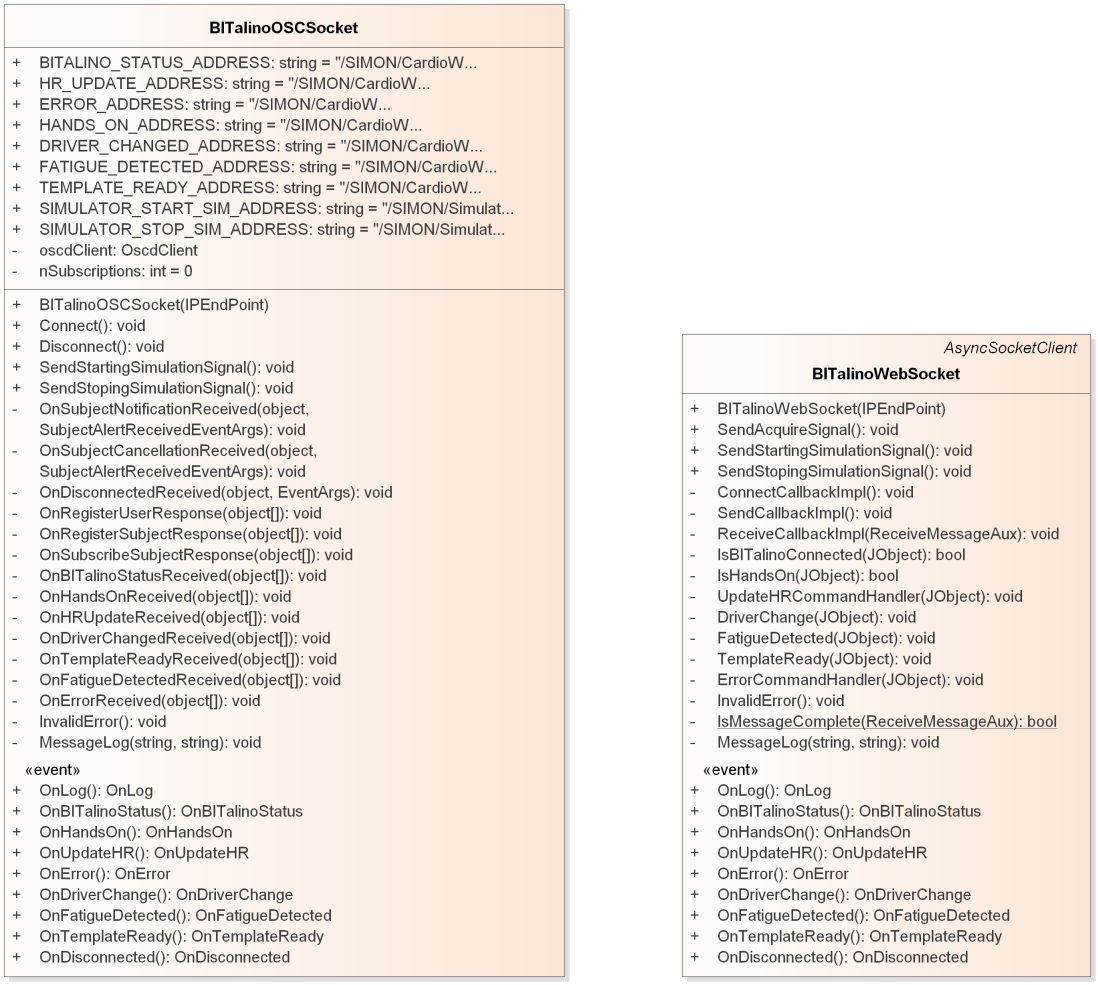
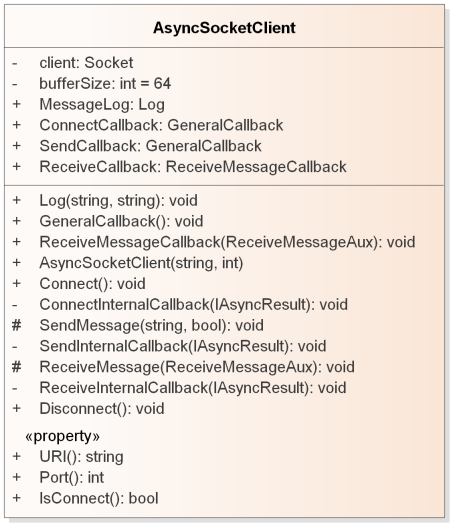


Figure F.2: `BITalinoClient` class

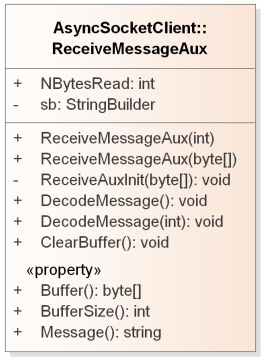


(a) `BITalinoOSCSocket` (b) `BITalinoWebSocket`

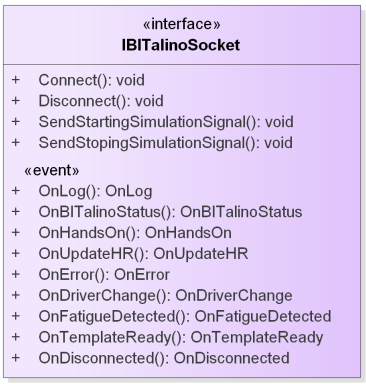
Figure F.3: *BITalino* socket classes



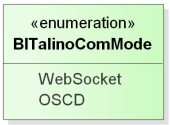
(a) `AsyncSocketClient`



(b) `ReceiveMessageAux`

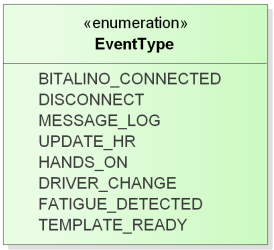


(c) `IBITalinoSocket`



(d) `BITalinoComMode`

(e) `BITalinoEvent`



(f) `EventType`

Figure F.4: Auxiliary class of the `BITalinoClient` package

G

`GameManager` Associated Classes

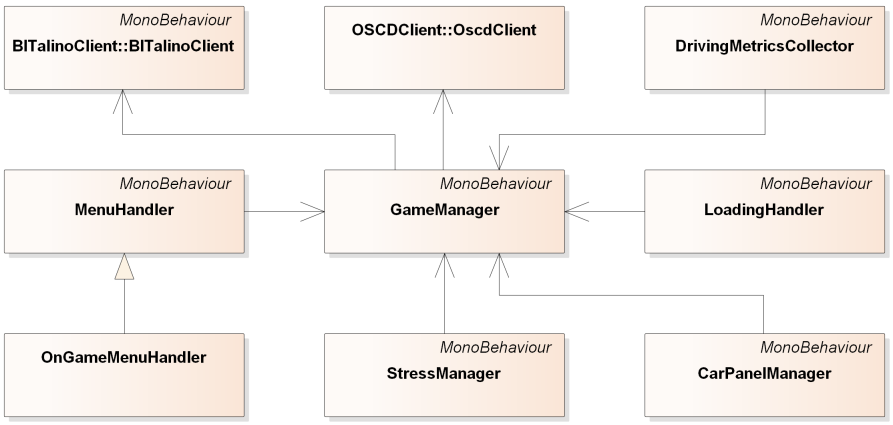


Figure G.1: (Simplified) Structure of the classes associated with the `GameManager` component

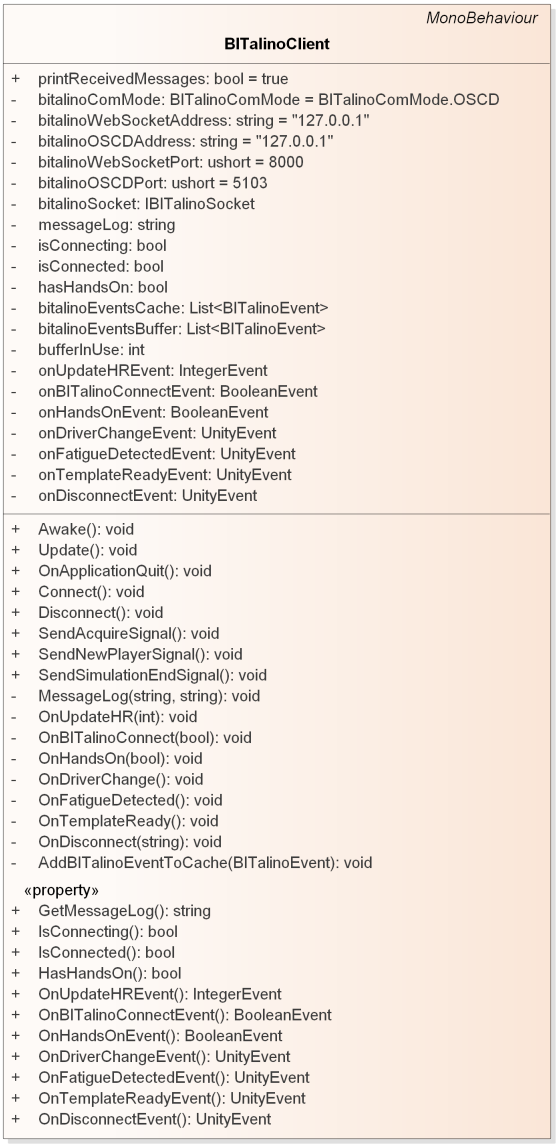
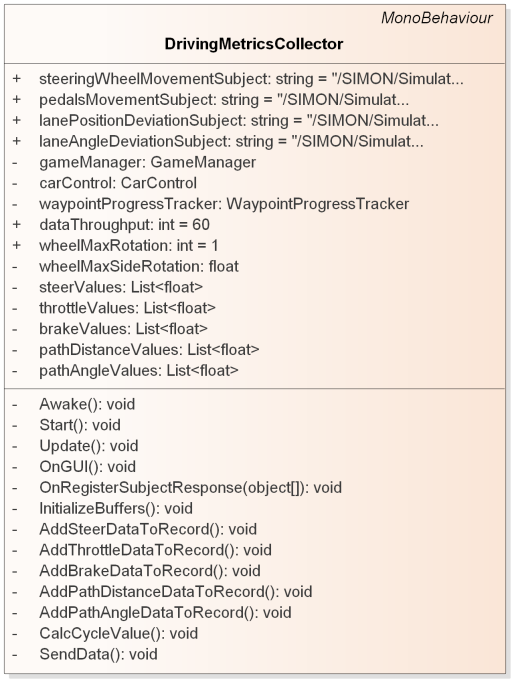
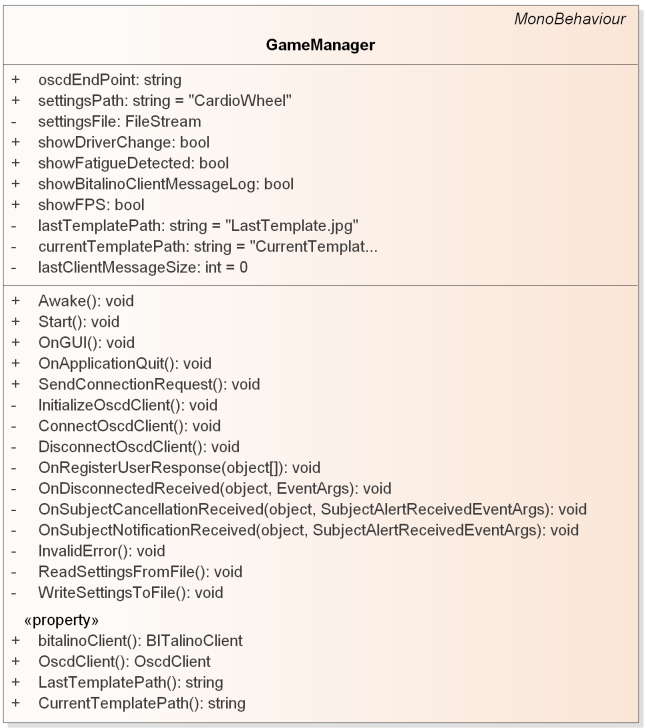


Figure G.2: `BITalinoClient` class

Figure G.3: `OscdClient` class

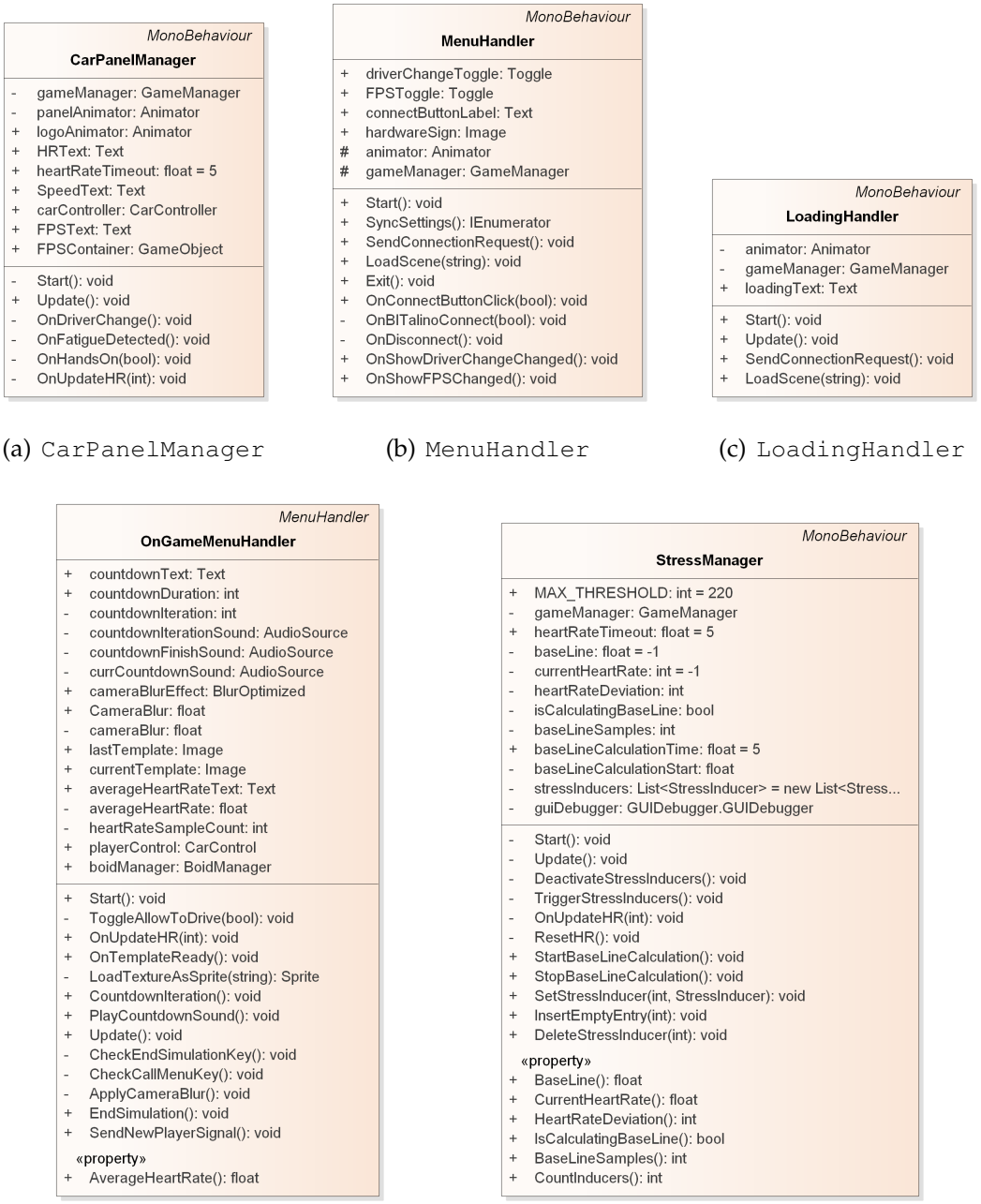


(a) `DrivingMetricsCollector`



(b) `GameManager`

Figure G.4: Classes associated with the `GameManager` component



(a) `CarPanelManager` (b) `MenuHandler` (c) `LoadingHandler`

(d) `OnGameMenuHandler` (e) `StressManager`

Figure G.5: Classes associated with the `GameManager` component

Simulator Experimental Evaluation

H.1 *Subject A*

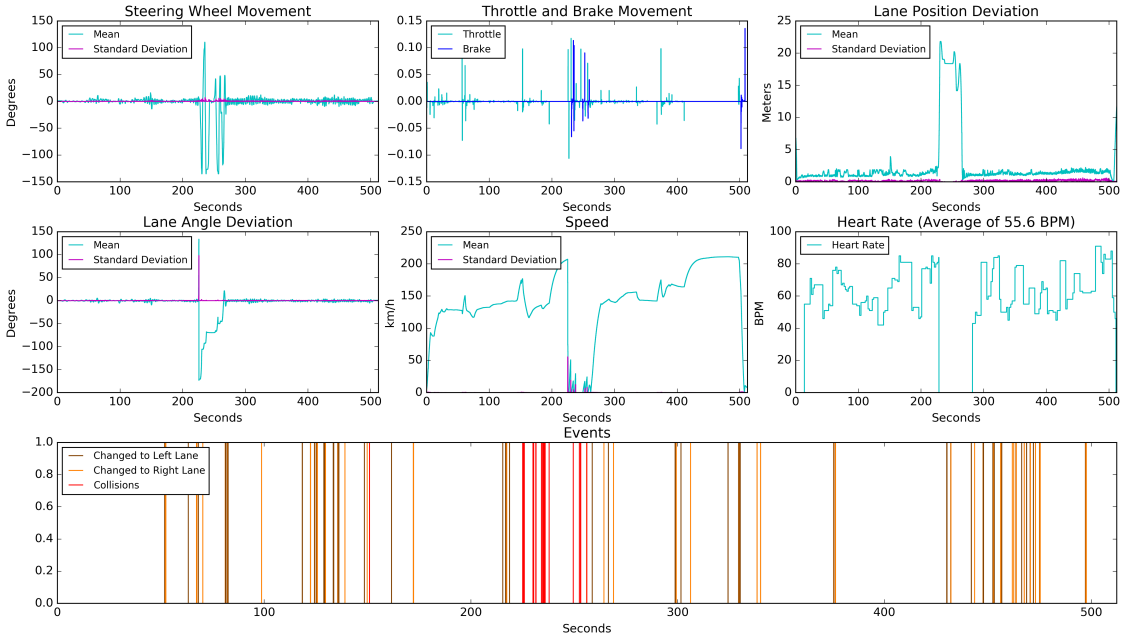


Figure H.1: *Subject A* - Performance graphs

H.2 *Subject B*

Before Alcohol

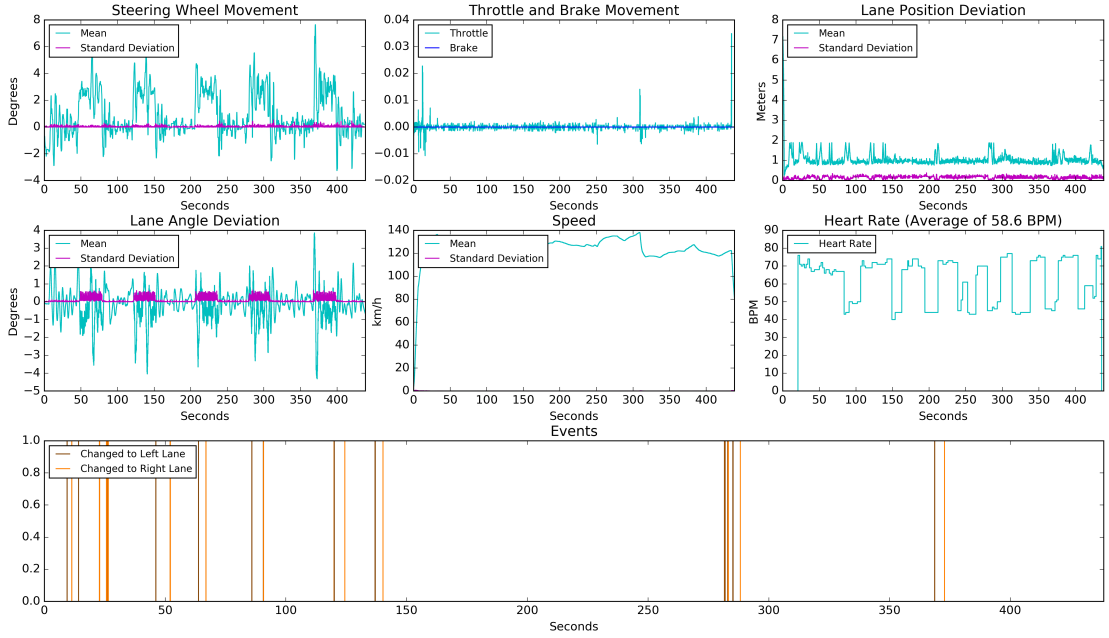


Figure H.2: *Subject B* - Performance graphs before alcohol consumption

After Alcohol

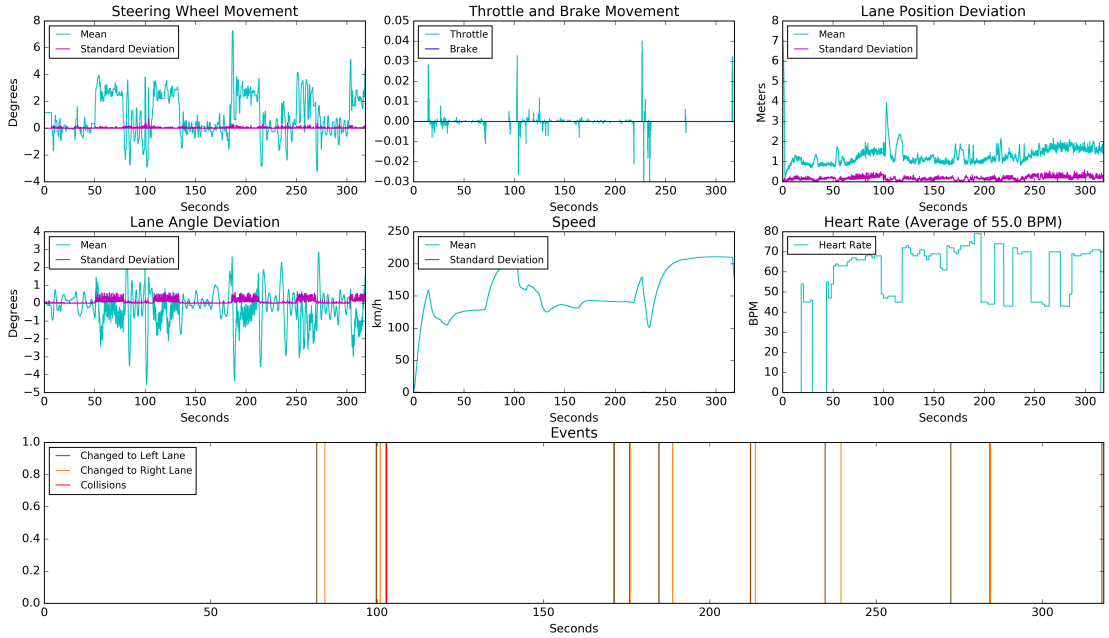


Figure H.3: *Subject B* - Performance graphs after alcohol consumption

H.3 *Subject C*

Before Alcohol

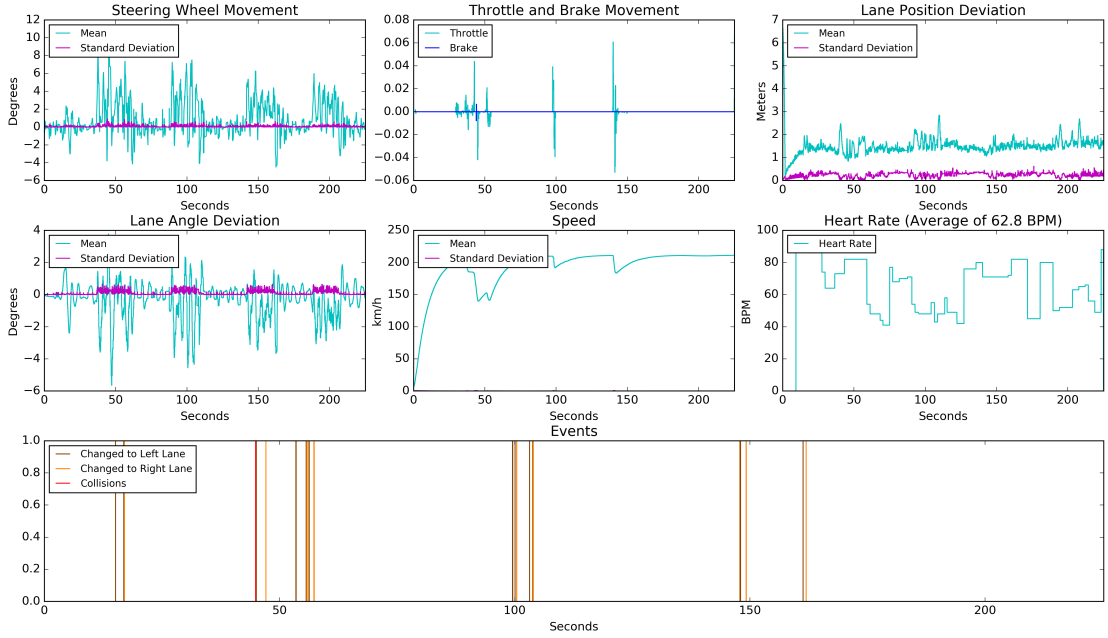


Figure H.4: *Subject C* - Performance graphs before alcohol consumption

After Alcohol

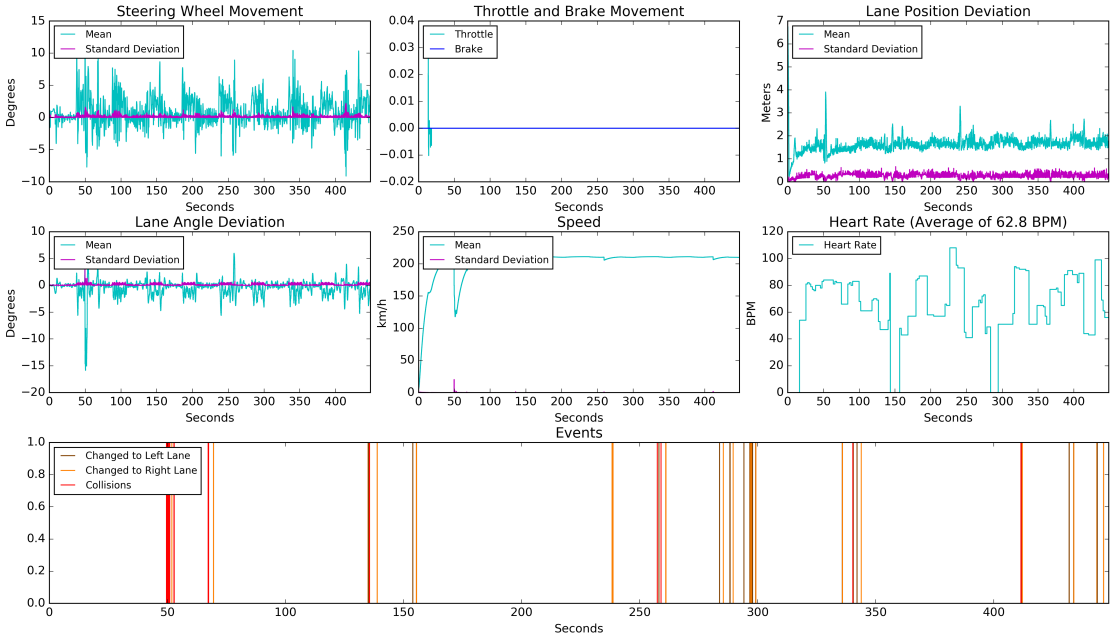


Figure H.5: *Subject C* - Performance graphs after alcohol consumption

H.4 *Subject D*

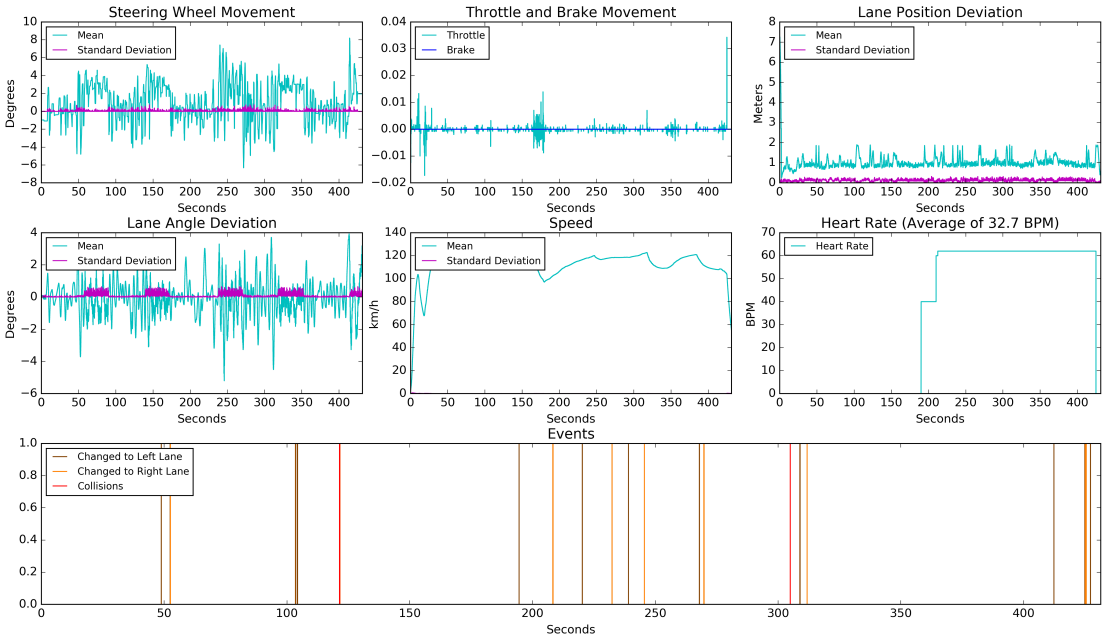


Figure H.6: *Subject D* - Performance graphs

H.5 *Subject E*

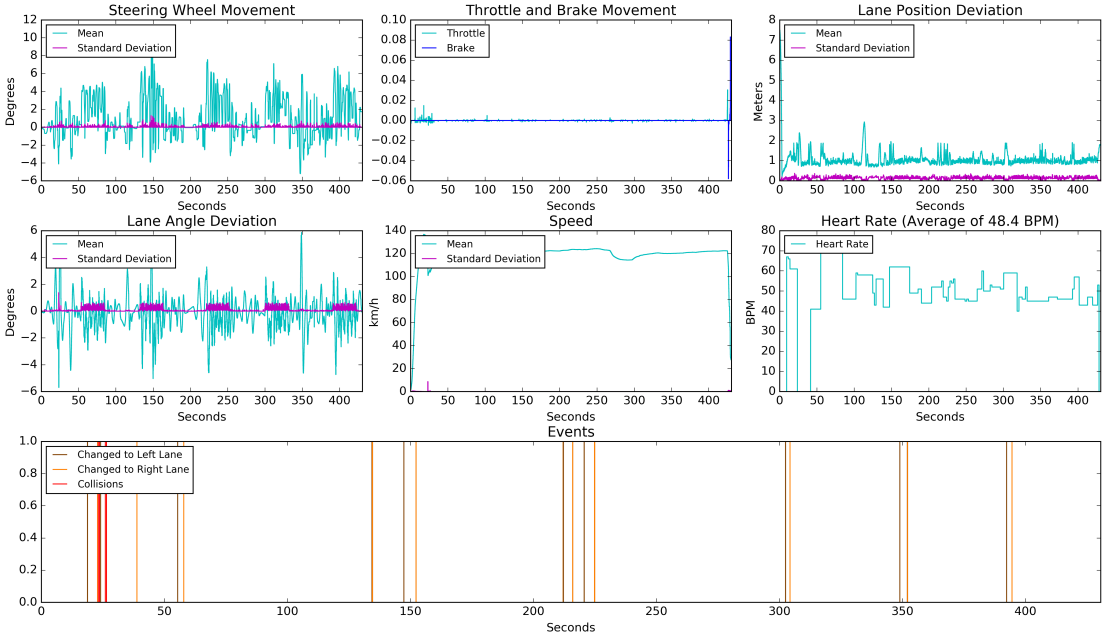


Figure H.7: *Subject E* - Performance graphs

H.6 *Subject F*

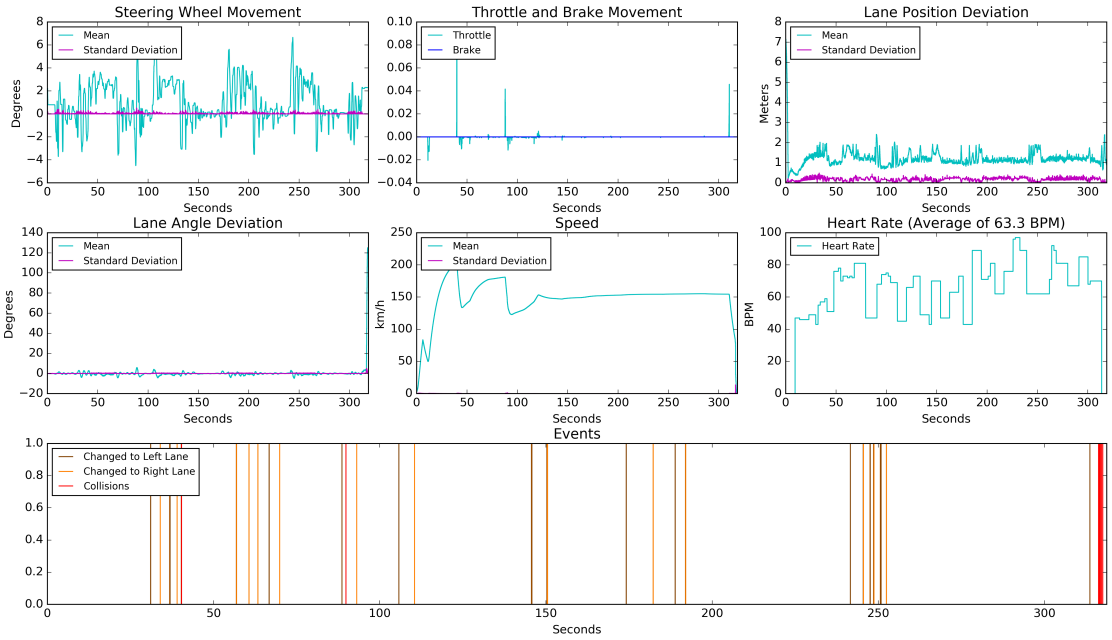


Figure H.8: *Subject F* - Performance graphs

H.7 *Subject G*

Before Alcohol

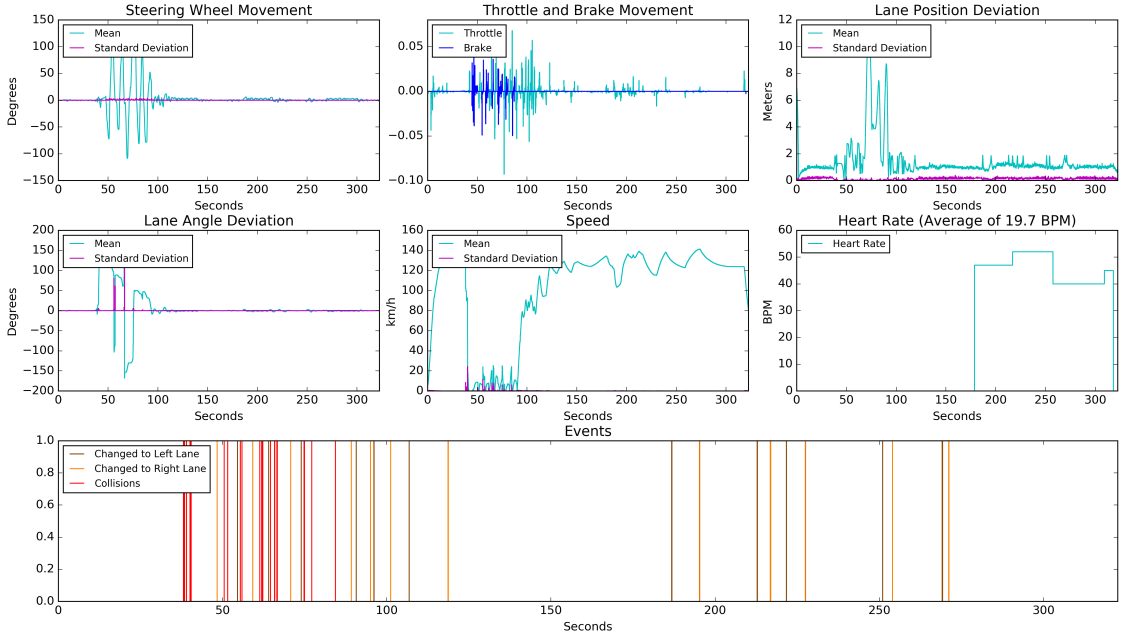


Figure H.9: *Subject G* - Performance graphs before alcohol consumption

After Alcohol

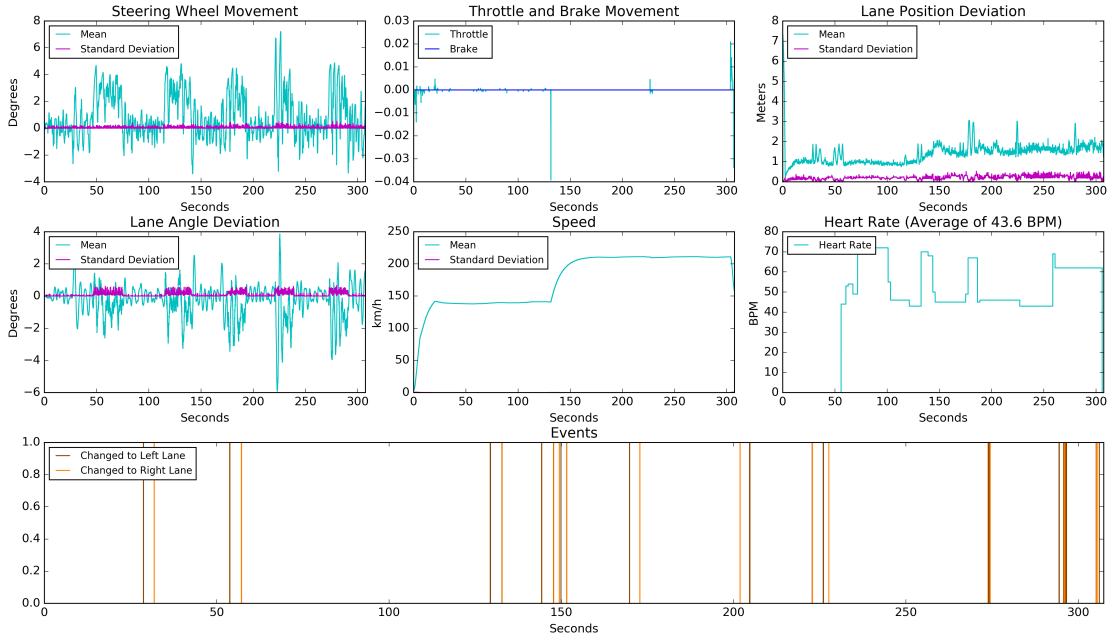


Figure H.10: *Subject G* - Performance graphs after alcohol consumption

H.8 *Subject H*

Before Alcohol

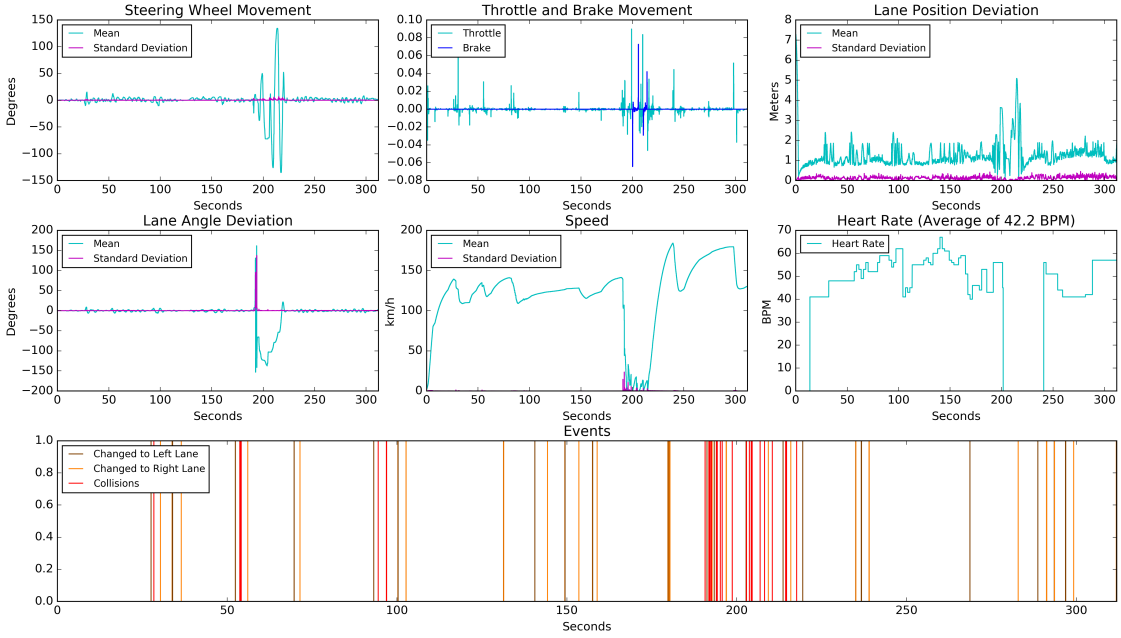


Figure H.11: *Subject H* - Performance graphs before alcohol consumption

After Alcohol

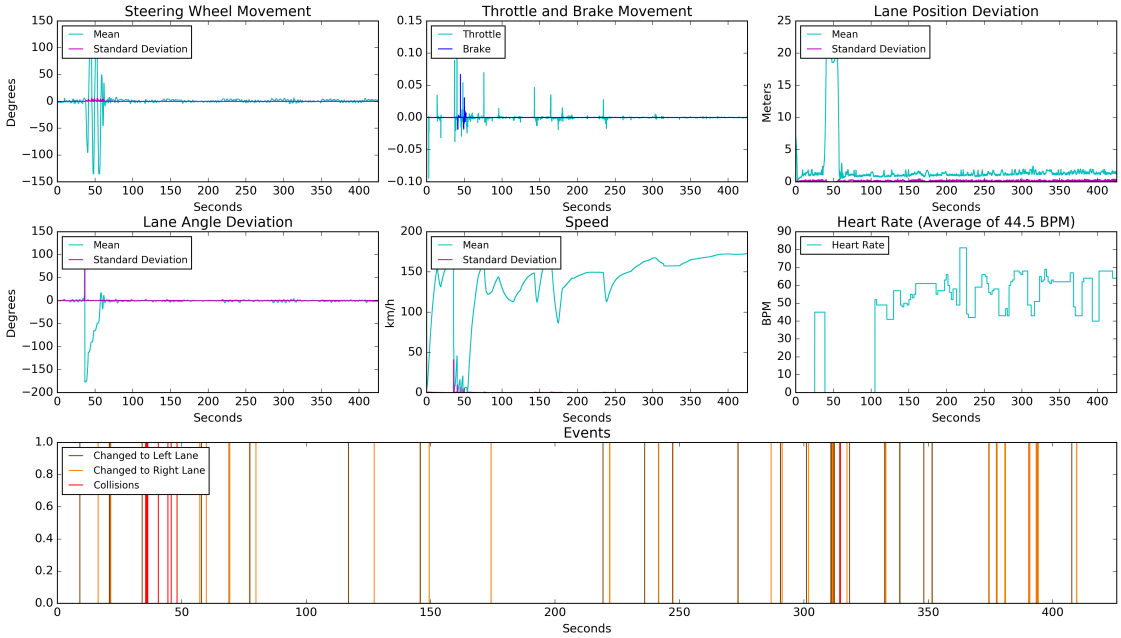


Figure H.12: *Subject H* - Performance graphs after alcohol consumption

H.9 *Subject I*

Before Alcohol

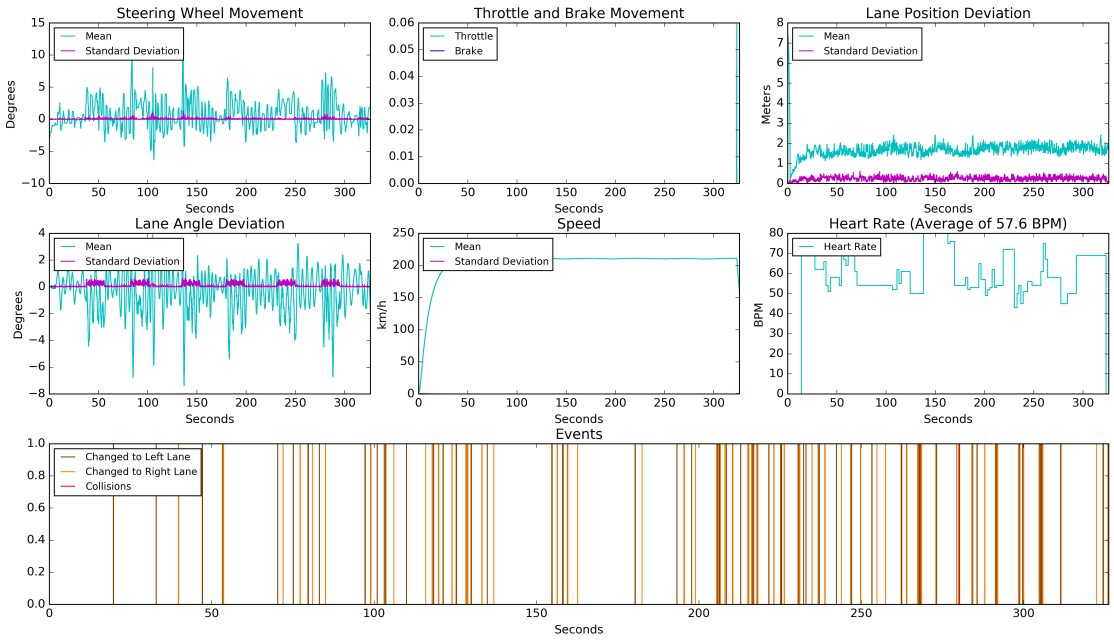


Figure H.13: *Subject I* - Performance graphs before alcohol consumption

After Alcohol

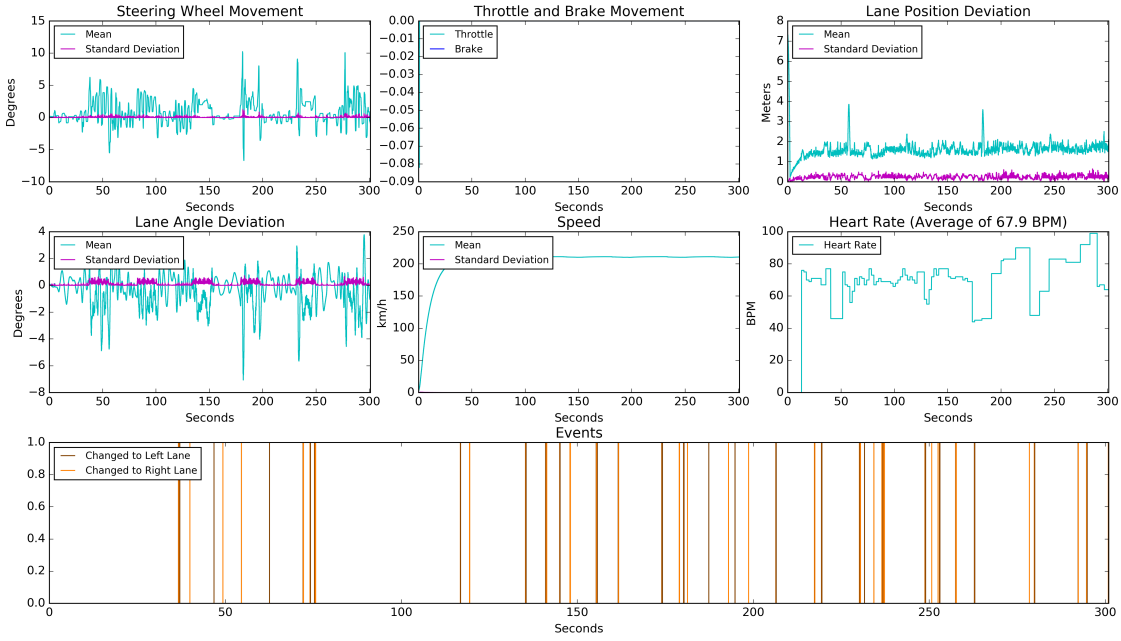


Figure H.14: *Subject I* - Performance graphs after alcohol consumption

H.10 *Subject J*

Before Alcohol

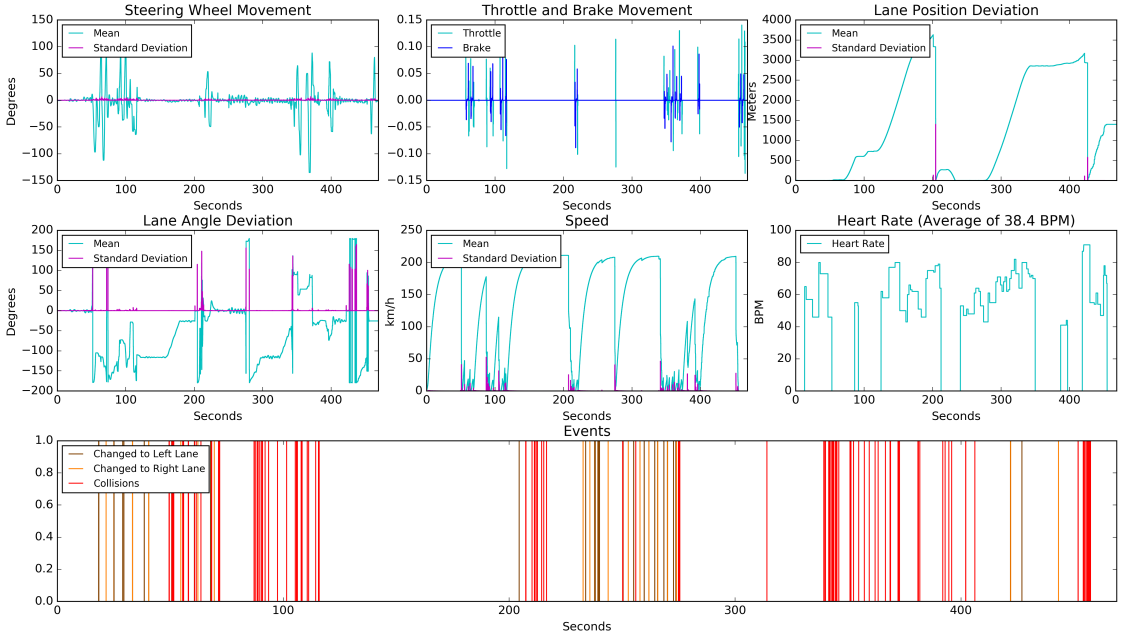


Figure H.15: *Subject J* - Performance graphs before alcohol consumption

After Alcohol

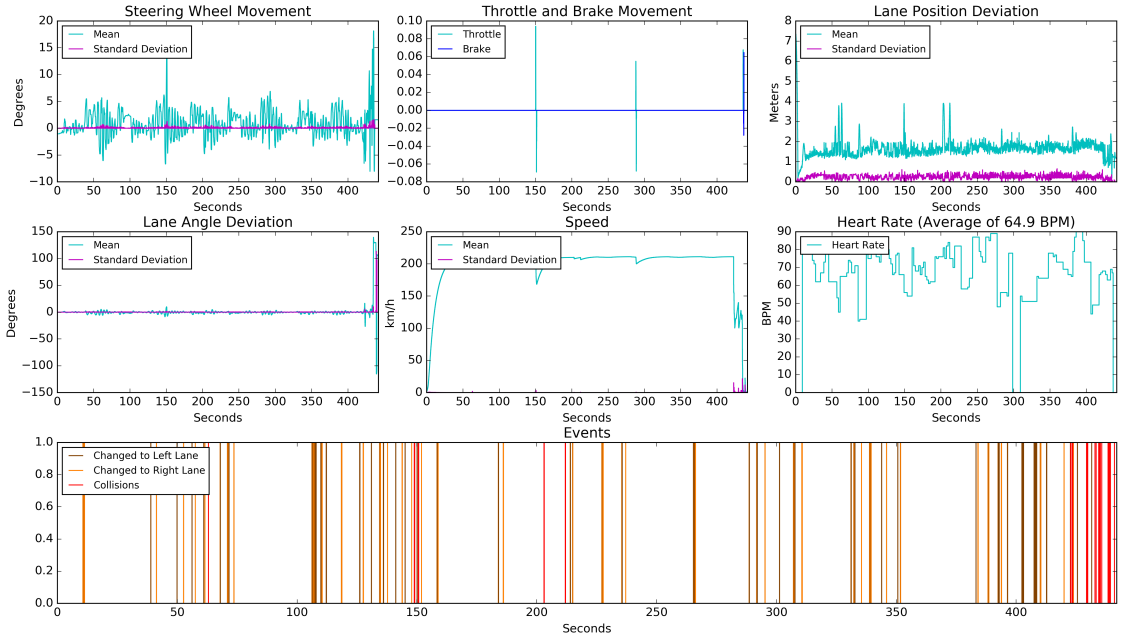


Figure H.16: *Subject J* - Performance graphs after alcohol consumption