FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Racing dashboard mobile application
Student:	Bc. Adam Gelatka
Supervisor:	Ing. Lukáš Hromadník
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of winter semester 2022/2023

Instructions

Design, implement and test an iOS application displaying data from a racing simulator. The application should be able to display vehicle information, such as current speed, engine revolutions, race information, such as the number of laps, current lap time, and advanced data such as the accelerator/brake pedal usage development graph during a lap.

1. Analyze existing applications used for displaying data from gaming simulators.
2. Analyze and describe a data structure provided by a chosen racing simulator.
3. In accordance with the supervisor, specify the functional and non-functional requirements of the application.
4. In accordance with the supervisor, design a graphical user interface of the application.
5. Design and implement the networking layer that will the application use to receive data from racing simulators.
6. Implement the mobile application as specified.
7. Test the mobile application.
8. Summarize the results of the work, describe its benefits.

Electronically approved by Ing. Michal Valenta, Ph.D. on 22 June 2021 in Prague.

Master's thesis

Mobile Application Racing Dashboard

Bc. Adam Gelatka

Department of software engineering
Supervisor: Ing. Lukáš Hromadník

May 5, 2022

Acknowledgements

I would first like to thank my thesis advisor and mentor, Ing. Lukáš Hromadník, whose expertise was invaluable and always devotedly helped whenever I ran into a trouble spot or had a question about my thesis. In addition, I would like to thank my parents for their wise counsel and sympathetic ear.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 5, 2022 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2022 Adam Gelatka. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Gelatka, Adam. *Mobile Application Racing Dashboard.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstract

This thesis is concerned with a mobile application that simulates a virtual vehicle dashboard by displaying data from racing simulator games. The application, which has been designed and implemented to run on the iOS platform, allows one to display relevant data about the vehicle, such as speed, engine revolutions, pedal usage, etc. In addition, time, position, and other race-related information is also displayed. UDP protocol is used for communication with various racing simulators. Statistical data from every session are recorded and presented using graphs that can be reviewed later.

Keywords racing simulator, car dashboard, mobile application, iOS, Swift, UDP

Abstrakt

Práce se zaobírá mobilní aplikací, která simuluje palubní desku automobilu zobrazující data ze závodních simulátorů. Navržená a implementovaná aplikace běžící na iOS platformě umožňuje zobrazovat relevantní data ohledně vozidla, např. rychlost, otáčky motoru, využití pedálů a další. Dále jsou viditelná data o času, pozici apod. Pro komunikaci se simulátory je používán UDP protokol, přičemž z každého závodu jsou ukládána statistická data, která jsou následně prezentována pomocí grafů.

Klíčová slova závodní simulátor, palubní deska, mobilní aplikace, iOS, Swift, UDP

Contents

Introduction **1**

1 Goals and Requirements **1**
1.1 Application Requirements 1
1.1.1 Functional Requirements 2
1.1.2 Non-functional Requirements 3
1.1.3 Use Cases 3

2 Analysis **5**
2.1 Racing Simulators 5
2.1.1 Market Share 6
2.2 Existing Alternatives 6
2.2.1 RealDash 7
2.2.2 Race Dash for Sim Games 8
2.2.3 DashPanel 10
2.2.4 Comparison 11
2.3 Data Output 12
2.3.1 UDP 13
2.4 Target Platform 14
2.4.1 iPhone 14
2.4.2 iPad 15
2.5 Data Packet Structure 15

3 Used Technologies **21**
3.1 GUI Creation Methods in iOS 21
3.2 Swift 22
3.3 Xcode IDE 22
3.4 Cocoapods 23
3.5 Swift Package Manager 23

3.6 Frameworks 23
3.6.1 UIKit 24
3.6.2 SnapKit 24
3.6.3 Swinject 24
3.6.4 Combine 24
3.6.5 SwiftNIO 25
3.6.6 CoreData 25
3.6.7 Charts 25
3.6.8 XCTest 26
3.6.9 SwiftLint 26

4 Design **27**
4.1 Architecture 27
4.1.1 MVVM 27
4.1.2 MVVM in iOS 28
4.1.3 Flow Coordinators 28
4.2 Mobile First Design 29
4.3 Separation of Concerns 29
4.4 Dependency Injection 29
4.5 Localization 30
4.6 Graphical User Interface 30
4.6.1 Main Menu Screen 30
4.6.2 Graphs Screen 31
4.6.3 Connection Screen 32
4.6.4 Dashboard Screen 34
4.6.5 Application Icon 35
4.7 Race Statistics Database 36
4.8 Common Data Packet 37
4.9 Networking Layer 38

5 Implementation **41**
5.0.1 Networking Service 41
5.0.2 Channel Handler 42
5.0.3 Data Decoder 43
5.1 Application Logic 44
5.1.1 Connection Process 44
5.1.2 Data Recording 45
5.2 Architecture 46
5.3 User Interface 47
5.3.1 Vertical Selector 47
5.3.2 Gradient Button 48
5.3.3 Panel Card 48
5.3.4 Progress Indicator 48
5.3.5 Bar Gauge 48

5.3.6 RPM Gauge . . . 49
5.3.7 Other Elements . . . 49
5.3.8 Documentation . . . 49

6 Testing **51**
6.1 Performance Testing . . . 51
6.1.1 Testing Environment . . . 51
6.1.2 Testing Results . . . 51
6.2 Unit Testing . . . 52
6.3 GUI Testing . . . 53
6.4 Usability Testing . . . 53
6.4.1 Testing Procedure . . . 54
6.4.2 Results . . . 55
6.5 Practical Testing . . . 56

Conclusion **57**

Bibliography **59**

A Acronyms **63**

B Contents of Enclosed CD **65**

C Racing Dashboard Screenshots **67**

List of Figures

1.1 Use case diagram 4

2.1 RealDash menu screen 8
2.2 RealDash dashboard screen 8
2.3 Race Dash menu screen 9
2.4 Race Dash dashboard screen 10
2.5 DashPanel menu screen 11
2.6 DashPanel dashboard screen 11
2.7 Encapsulation of UDP 13
2.8 UDP header 14

4.1 Model-View-ViewModel diagram 28
4.2 Model-View-ViewModel in iOS diagram 28
4.3 High-fidelity prototype of main menu 32
4.4 High-fidelity prototype of graphs screen 33
4.5 High-fidelity prototype of the connection screen 34
4.6 High-fidelity prototype of modern dashboard 36
4.7 Application icon 36
4.8 Conceptual diagram of race records database 37

6.1 Practical testing in Forza Horizon 5 56

C.1 Main menu screen screenshot 67
C.2 Graph screen screenshot 68
C.3 Connection screen screenshot 68
C.4 Connection screen guided screenshot 69
C.5 Dashboard screen screenshot 69

List of Tables

2.1 Racing games examples 5
2.2 Racing games market research report by Market Research Future . 6
2.3 Racing games market research report by Statista 6
2.4 Comparison of existing applications 12
2.5 Communication protocols of racing games 12

6.1 Networking performance testing 52
6.2 Usability testing users 54

List of Source Code Listings

1 First part of the FH5 data structure 16
2 Second part of the FH5 data structure 17
3 Third part of the FH5 data structure 18
4 Common data structure 39
5 Networking service protocol 42
6 Networking service stop function 42
7 Networking service start function 43
8 Networking service handler 44
9 Networking service decoder 45
10 Coordinator protocol 47
11 Unit tests example 53

Introduction

Racing simulators have become a popular genre in the video game industry. The essence of racing simulators is to provide an almost real driving experience. Simulators are generally designed to simulate real-world activities with an eye for accuracy. Racing simulators are no exception, based on variables such as tire grip, fuel consumption, chassis yaw, gravity, friction, etc. Players are not only individuals seeking enjoyment, but also professional racing drivers perfecting their skills without the need for a real car and race track. A simple simulator setup could consist of only a computer or console, a monitor, and a gamepad. To further improve the experience, additional external devices can be used, for example, a steering wheel, pedals, gear shifter, handbrake, and even external dashboards.

This master thesis follows the concept of an external dashboard by designing and implementing a mobile application that serves as a virtual dashboard for various racing simulators, such as Assetto Corsa and arcade-styled Forza Horizon 5. In-game HUDs can negatively impact drivers' visibility by taking part in the screen. The motivation behind the Racing Dashboard application is to decrease the interference between the field of view and the HUD by placing the dashboard wherever needed as an external component.
Almost all racing simulators support a data-out feature, which tends to be often based on UDP networking protocol. For the Racing Dashboard application, a communication layer is designed based on the documentation analysis of data-out features of various racing simulators. In addition to the communication component, a full-fledged application with a modern user interface is designed, implemented, and fine-tuned to offer a seamless experience. Among other modern technologies, a reactive programming paradigm is used to ensure a sustainable data flow throughout the entire application. The Racing Dashboard records the data for each session and presents them in graphs as a statistical resource that can be reviewed by the user.

At the beginning of the thesis, it is necessary to analyze all the technologies that will be used to implement the mobile application for the iOS

platform, namely for iPhone and iPad devices. The design chapter deals with the application architecture, design patterns, and the GUI design process itself. The implementation section describes the implementation process that adapts the knowledge gathered from previous sections. The finished application is tested, including a practical test on a racing simulator, and, in conclusion, the thesis is evaluated.

Chapter 1

Goals and Requirements

The main objective of this thesis is to design and implement a mobile application that acts as a virtual racing dashboard. The purpose of the dashboard is to display relevant information about the vehicle and the race itself. The application is intended for the iOS platform, specifically for iPhone and iPad devices.

The initial step is to thoroughly analyze existing dashboard applications and record their limitations, pros, and cons. The design, implementation, and testing process is well documented and evaluated. The result is a fully functional networking component that is versatile, well-performing, and covers the communication between a variety of racing simulators.

GUI design will emerge within the next goal to achieve a modern and lightweight interface. It is fundamental to make all application components well organized, intuitive, and non-disruptive.

An application written using the Swift programming language encapsulates the project and adapts the GUI design. Performance testing is necessary due to the potentially high data flow from the simulators. The persistence of data records will be ensured by using the CoreData framework. Finally, the finished application will be tested on a racing simulator.

1.1 Application Requirements

The application satisfies two sets of requirements. Some are based entirely on the assignment of the thesis, such as displaying speed and saving data to be presented as graphs. Other requirements originated from the analysis of existing dashboard applications and even from personal experience with racing simulators. The application will try to comply with Apple's Human Interface Guidelines [1]. Lastly, the app is designed similarly to a production-grade iOS application. Functional and non-functional requirements are stated in the subsequent sections.

1.1.1 Functional Requirements

Functional requirements are features or functions that developers must implement to allow users to perform given tasks. They generally describe the intended behavior of the application. Functional requirement analysis is a fundamental step in the software design process and has an immense impact on the finished product. In agreement with the supervisor, the following functional requirements were stated:

- **FR1: The application displays vehicle-related data:**
  - Vehicle's speed
  - Engine's revolutions per minute (RPM)
  - Accelerator, clutch, and brake pedal usage
  - Tire temperature
  - Fuel tank level
- **FR2: Application displays race-related data:**
  - Lap count
  - Current position
  - Total time
  - Best lap time
- **FR3: Recording data and saving statistics:** The application supports data collection and generation of statistics presented in graphs. Variables such as pedal usage, final position, best lap, and total time are considered to be eligible sources. All records will be persistently saved.
- **FR4: Multiple dashboard designs:** Multiple and easily switchable dashboard designs are featured. In the current state, the user does not have to make any further customization of the dashboard. The goal is to create one fully functional dashboard for the release in the hope of adding more in the future.
- **FR5: Connection guide:** A connection wizard guides the user through the connection process. The steps given are exclusive to every simulator, and the wizard must help the user to setup the connection. Partial automatization of the entire process is essential.

1.1.2 Non-functional Requirements

Non-functional requirements describe the general capabilities and characteristics of the application.

- **NFR1: Application for iOS platform:** The application is designed and built for the Apple iOS platform, specifically the for iPhone and iPad devices. The iOS 15 is the supported version with no further backward compatibility. The application does not support the MacOS or Apple Watch platform.
- **NFR2: Network data transfer:** The application supports networking using the UDP protocol used for communication between the application and the racing simulator of choice. Data are passed through a local network to which the device connects using Wi-Fi and acts as a UDP server.
- **NFR3: Data persistence:** The recorded data used as race statistics are persistently saved in the device's local storage. Currently, no cloud solution is supported.
- **NFR4: Intuitive GUI:** Modern and intuitive GUI is mandatory, not only for the dashboard itself, but also for other supporting screens such as the main menu or the statistics overview. GUI is loosely adapting the Apple's Human Interface Guidelines [1]. Responsivity of the GUI is mandatory due to the support of different screen sizes.

1.1.3 Use Cases

Use cases define how users will perform tasks in the application. Each use case is represented as a scenario that outlines the behavior of the application.

- **UC1: Launching the dashboard:**
  - The user opens the application.
  - The user selects a racing simulator.
  - The user selects a dashboard type.
  - The user starts the connection process.
  - Once connected, the user sees the connection details.
  - The user continues to the dashboard.
- **UC2: Opening the connection guide:**
  - The user opens the application.
  - The user selects a racing simulator.

  - The user starts the connection process.
  - While connecting, the user sees the connection guide.
  - The user is able to swipe through the steps of the guide gallery.

- **UC3: Reviewing race statistics:**
  - The user opens the application.
  - The user taps the graphs button.
  - The user selects a given record using the navigation buttons.
  - The user selects a value to be displayed.
  - The user interacts with the graph.

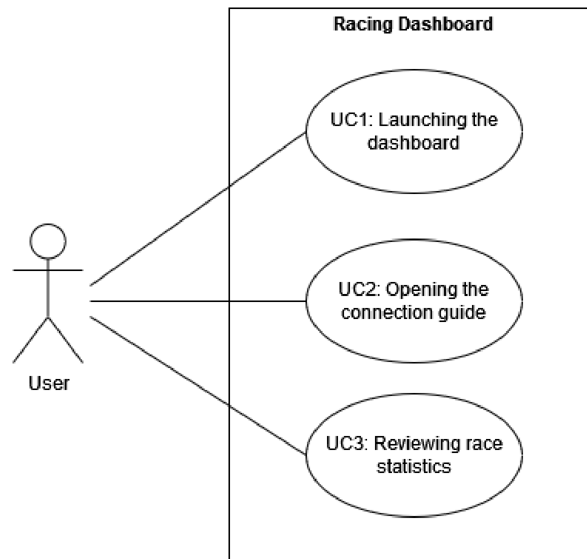

Figure 1.1: Use case diagram

Chapter 2

Analysis

2.1 Racing Simulators

The topic of this thesis revolves around racing simulators, which are essentially video games developed mainly for computers and video game consoles. There are even some racing simulators designed for mobile platforms, but these will not be taken into account within the scope of this project. All simulators have one thing in common, the desire for realism. Their goal is to simulate real-world behavior based on different variables such as friction, tire grip, fuel level, weight, aerodynamics, etc. Advanced simulators allow users to modify settings such as gear ratio, suspension behavior, ride height, and many more. Today, modern racing games require a decent amount of processing power, mainly due to the advanced graphics. The most famous simulators are Assetto Corsa, Project CARS, F1 Career Challenge, and more. The arcade-styled side of the genre is, for example, the Forza Horizon series or The Crew. Arcade styled games are not entirely focused on realism, featuring elements such as events or quests. An aggregation of both styles is called simcade, and a well-known installment is the Forza Motorsport series or Gran Turismo.

Table 2.1: Racing games examples

Racing game	**Type**	**Year released**
Assetto Corsa	simulator	2014
Forza Horizon 5	arcade	2021
Project CARS	simulator	2015
Forza Motorsport 7	simcade	2017
F1 2021	simulator	2021
Dirt Rally	simulator	2015

2.1.1 Market Share

As stated in the Introduction, racing games form a significant share of the video game industry. The prospects for this genre are very positive. As can be seen in Table 2.2, Market Research Future [2] predicts a compound annual growth rate of 11.6% by 2030. The market share for the year 2019 is reported to be $1,364 million. The other statistical source shown in table 2.3, Statista [3], projects a growth of 10.31% between the years 2022-2026. On the other hand, Statista reports a significantly higher market share for 2022, projected to be $3,227 million. It is evident that both research reports are quite distinct; however, the expected trend is a common denominator for both. It is projected to see a growth rate of around 10% within the next 10 years. An overview of both reports can be seen in Tables 2.2 and 2.3.

Table 2.2: Racing games market research report by Market Research Future[2]

Report attribute/metric	**Details**
Market size	$1,364m
CAGR	11.6%
Base year	2019
Forecast period	2020-2030

Table 2.3: Racing games market research report by Statista[3]

Report attribute/metric	**Details**
Market Size	$3,207m
CAGR	10.31%
Base year	2022
Forecast period	2022-2026
Projected market volume 2026	$4,749m

2.2 Existing Alternatives

The mobile application market is saturated with virtual dashboard applications. The following analysis was performed on applications for the iOS platform available on the Apple App Store. Various aspects were considered, such as the quality of the GUI, the number of supported games, dashboards, rating, supported iOS versions, and many more. The most famous applications are the following:

- Real Dash
- Race Dash for Sim Games

- DashPanel
- Sim Racing Telemetry
- RS Dash
- pCars Dash

The in-depth analysis of the first 3 applications is documented in the following section, including personal insights and a comparison table as a part of the conclusion.

2.2.1 RealDash

RealDash is one of the most famous applications for virtual dashboard telemetry. The GUI tends to be modern; however, it might be confusing at some point. Among the supported games is Assetto Corsa, BeamNG Drive, Dirt Rally, Forza Horizon 4, Forza Motorsport 7, and many more. In total, the application currently supports up to 10 different racing simulators [4]. Furthermore, RealDash even enables data input from real cars. The list of supported ECU models may differ between the iOS, Android, and Windows 10 versions of the application. The iOS application supports ECUs such as Autronic SM4, EasyEcu 3+, Ecumaster EMU series, Speeduino, Megasquirt, and many more [4]. This addition allows to display data from a real vehicle that is equipped with an OBD2 port. However, a compatible OBD2 Wi-Fi adapter device is required.

RealDash offers multiple free and paid dashboard designs. The application comes with some free basic dashboards that feature a rather poor design. However, paid dashboards are visually more elaborate and appear to be modern. Dashboards are highly customizable in different ways, such as changing the displayed gauges, positions, colors, and many more. The GUI of the application seems to be not optimized in some places for newer iPhones with a notch. There is also a menu that is based on a 3D car model. The user can interact with the model to modify settings such as tire width, engine size, gear ratios, and many more. This approach might be confusing to some users, due to the fact that the application is mixing flat modern design with 3D objects. RealDash also offers a paid version of the application that removes several limits, ads, and unlocks functionalities such as data logging. The application supports iPhone and iPad devices. The development of RealDash continues to this day. RealDash screenshots are shown in figures 2.1 and 2.2.

- **Developer:** Napko Oy
- **Rating on App Store:** 5.0/5 (2 ratings)
- **Supported iOS:** iOS 8.0 or later

- **Free/Paid:** Both
- **Additional dashboards:** Paid

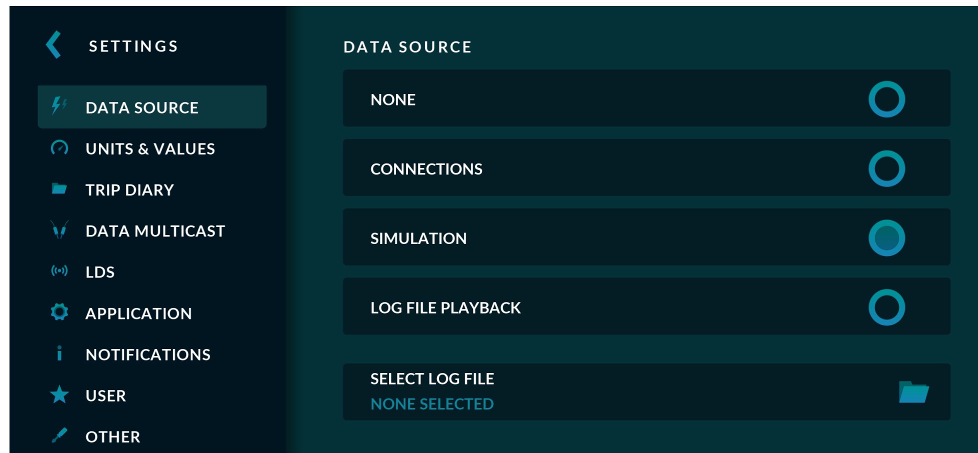

Figure 2.1: RealDash menu screen

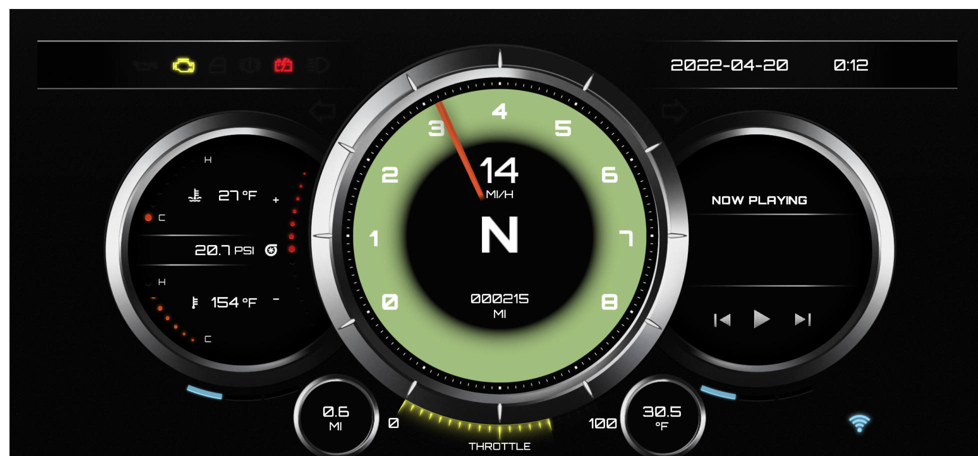

Figure 2.2: RealDash dashboard screen

2.2.2 Race Dash for Sim Games

Race Dash for Sim Games is a another representative of racing dashboard applications. It supports racing games such as Forza Horizon 4/5, F1 2021, Assetto Corsa, PCars2, and more. In total, 11 games are supported [5].

The application tends to have a simple and flat design. The menu is not very intuitive, although at some point it is easy to use. Race Dash offers some dashboards for free, although others are unlocked by paying a permanent unlock, or a subscription. Dashboards offer slight customization that allows to change properties such as background color. Furthermore, the settings menu allows one to change parameters such as measurement units, UDP port, and many more preferences. In addition to the support of iPhone and iPad devices, Race Dash also offers an Apple Watch application, which can display a small dashboard with the most basic values. Race Dash does not support data recording. GUI seems to be unoptimized for newer iPhones with a notch. The development continues to this day. Screenshots of Race Dash are shown in figures 2.3 and 2.4.

- **Developer:** David Mills
- **Rating on App Store:** 4.1/5 (7 ratings)
- **Supported iOS:** iOS 9.0 or later
- **Free/Paid:** Both
- **Additional dashboards:** Paid

Figure 2.3: Race Dash menu screen

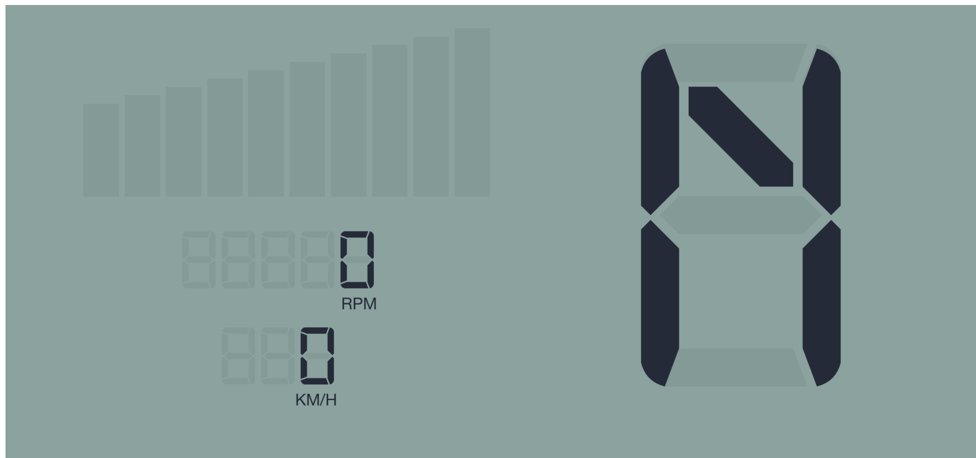

Figure 2.4: Race Dash dashboard screen

2.2.3 DashPanel

DashPanel is the last example of the listed application alternatives. Due to the loading screen, it is clear, that the application is developed using Unity and does not blend very well with the iOS environment. DashPanel supports racing games such as Assetto Corsa, Forza Horizon 4, F1 2021, PCars2, and more. In total, the application supports up to 9 games. The design of the main menu is very outdated, although it is easy to use. Within the settings, the user can change measurement units, UDP parameters, and other properties. DashPanel features fully customizable dashboards that can be created and modified using a built-in editor. Predefined dashboards are also available with the addition of community content. All featured dashboards come free with the application; however, the application requires transactions for each racing game in order to unlock full data displaying capability; moreover, the application does not support data logging. The interface is outdated, unintuitive, and unoptimized for newer iPhones with a notch. DashPanel runs on iPhone and iPad devices. The development continues to this day. Screenshots of the Dash Panel are shown in figures 2.5 and 2.6.

- **Developer:** Bernhard Deininger
- **Rating on App Store:** 3.7/5 (3 ratings)
- **Supported iOS:** iOS 9.0 or later
- **Free/Paid:** Both
- **Additional dashboards:** Paid

Figure 2.5: DashPanel menu screen

Figure 2.6: DashPanel dashboard screen

2.2.4 Comparison

The results of the analysis are summarized in the comparison table listed below 2.4. The table shows the number of supported games and a personal opinion about the GUI on a given scale: *Weak*, *Neutral*, *Good*, where *Weak* represents the weakest GUI designs and conversely *Good* represents the best GUI designs. In addition, an App Store rating and a purchase plan are included.

As can be seen in Table 2.4, almost all applications suffer from a rather poor user interface. Most of them support up to 10 games or even more. RealDash additionally supports data from real car ECUs and can be utilized

Table 2.4: Comparison of existing applications

Application name	Games	GUI	Plan	Rating
Real Dash	14	Neutral	Free/Paid	5/5
Race Dash	11	Neutral	Free/Paid	4.1/5
DashPanel	9	Weak	Free/Paid	3.7/5
Sim Racing Telemetry	10	Good	Trial/Paid	-
RS Dash	17	Weak	Paid	5/5
pCars Dash	1	Weak	Paid	5/5

within real races. However, all listed applications include paid content and, in some cases, at least one purchase is required to use the application. The paid content can be divided into three categories - paid dashboards, paid data unlocks, and paid premium accounts. Most dashboard applications support both iPhone and iPad devices, some of them even support the Apple Watch.

2.3 Data Output

Almost all racing simulators offer a data output feature, usually using shared memory or by sending data over the network. To decide which protocols the application should support, it is necessary to analyze the protocols implemented by racing simulators. The analysis took into account the following games: Asseto Corsa, Forza Horizon 5, Project CARS, Forza Motorsport 7, F1 2021, and Dirt Rally. The documentation of each listed game served as a data source for the analysis. The results are recorded in Table 2.5.

Table 2.5: Communication protocols of racing games

Racing game	Communication protocol
Assetto Corsa	UDP
Forza Horizon 5	UDP
Project CARS	UDP
Forza Motorsport 7	UDP
F1 2021	UDP
Dirt Rally	UDP

The results of the analysis point to the clear fact that the most widely used protocol for data transmission in racing simulators is UDP. Some of the games support shared memory output at the same time. The shared memory output method is based on a shared data file that is continuously updated by the game. However, shared memory is not usable in the case of this project, due to the wireless connection between the iOS device and the racing game. It is evident that the racing dashboard application will utilize

the UDP networking layer due to its frequent occurrence in the data output features of racing simulators.

2.3.1 UDP

User Datagram Protocol, in short, UDP, is a simple datagram-oriented transport layer protocol. Each operation creates a single UDP datagram that is being sent as an IP datagram. Unlike TCP/IP, in UDP reliability and duplicate protection are not guaranteed [6]. UDP should be avoided when reliable order and error-checked transmission is required. UDP can also be described as a scaled-down economy model of TCP; therefore, it is sometimes referred to as a thin protocol.

> ”Like a thin person on a park bench, a thin protocol doesn’t take up a lot of room”[7]

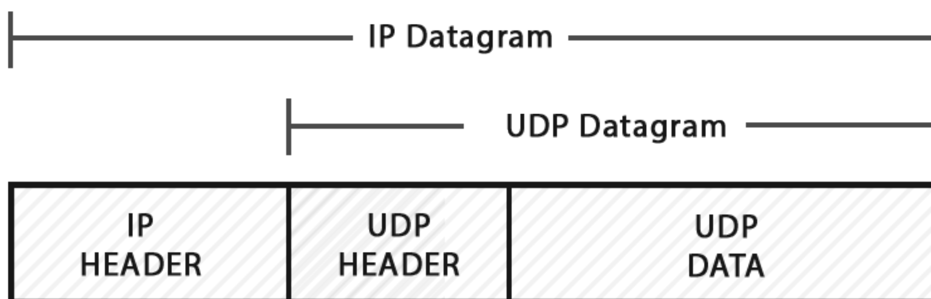

Figure 2.7: Encapsulation of UDP [6]

Figure 2.7 shows an encapsulated UDP datagram as an IP datagram. Ultimately, it is just a UDP datagram with the addition of an IP header. UDP datagram consists of the data itself and a UDP header. The structure of the UDP header can be seen in Figure 2.8. Port numbers are there to identify the sending and receiving process [8]. TCP and UDP port numbers are independent of the demultiplexing process. Next, the UDP length defines the length in bytes of the UDP header and the data itself. The minimum value of the length is 8 bytes. However, the length of the UDP datagram is redundant and can be calculated as the total length of the IP datagram subtracted by the length of the IP header [6].

Both TCP and UDP feature checksums are used to verify the integrity of the data by detecting potential errors that may occur. The IP header does have a checksum as well; however, it is derived from the header only. On the other hand, the UDP checksum takes into account both the header

and the data [6]. The network framework used in this project automatically handles the UDP checksum a reacts appropriately; therefore, no further description is needed.

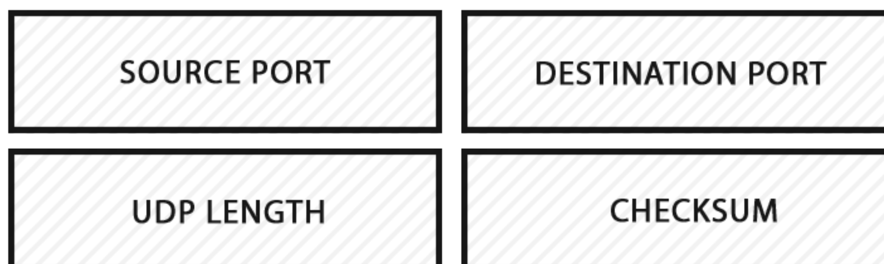

Figure 2.8: UDP header [6]

2.4 Target Platform

The target platform, as specified in the thesis assignment, is the iOS mobile operating system developed by Apple Inc. solely for its devices. The iOS is running on iPhone and iPad devices, although for iPads, iOS is branded as iPadOS. Both listed devices will be supported by the racing dashboard application.

As mentioned in the Introduction, the Racing Dashboard application will exclusively support the latest iOS version 15. This seemingly strict decision was made on the basis of the adoption data from Apple Inc. Apple claims that 72% of the iPhone devices introduced in the last four years (now is the year 2022) have iOS 15 installed. Furthermore, 63% of all iPhone devices run iOS 15. In the case of iPads, 57% of those introduced in the last four years have iOS 15 installed, and for all iPad devices, the number is 49% [9].

2.4.1 iPhone

The Apple iPhone is a line of smartphones with the first generation announced on January 9, 2007. It started as a combination of three products, a mobile phone, an iPod and an internet communication device [10]. The iPhone has developed substantially since its announcement and there is currently a 15th generation on the market [11]. The newest iOS version 15 supports 24 different iPhone models [12]. The biggest difference between the devices in terms of this project is the screen size, which the GUI has to adapt to. The device with the smallest screen across all the supported iPhones is the iPhone SE

(1st generation) with a 4-inch screen with a resolution of 1136 × 640 pixels [13]. On the contrary, the largest screen used in the iPhone 13 Pro Max is 6.7 inches diagonally with a resolution of 2778 × 1284 pixels [14].

2.4.2 iPad

The iPad is a tablet computer developed by Apple Inc. Originally, the iPad was supposed to ship before the iPhone. This did not happen, and the iPad was announced later on January 27, 2010. The first generation featured Wi-Fi and 3G (cellular) models [15]. Apple currently offers 4 different models of iPads, namely iPad Pro, iPad Air, iPad and iPad mini. The iPadOS 15 runs on 21 different iPad models [16]. The smallest screen within supported models is featured in the fourth generation of the iPad Mini with 7.9 inches diagonally and resolution of 2048 × 1536 pixels [17]. On the contrary, the largest iPad Pro has a 12.9-inch screen with resolution of 2732 × 2048 pixels [18].

2.5 Data Packet Structure

One of the key steps of this thesis is to analyze and describe a data structure provided by a chosen racing simulator. In terms of the racing simulator selection, Forza Horizon 5 was chosen, which is one of the newest simcade racing games. It is important to mention that the analyzed packet structure is the same for Forza Horizon 4 and nearly the same for Forza Motorsport 7. The wider usability of this specific data structure is the main reason why it was chosen. The description of the structure can be found on the Forza Motorsport forum [19]. Forza's data output feature is based on sending packets of the same format multiple times per second; to be more precise, the output rate is up to 60 packets per second, depending on the network quality. The total memory size of each packet is 311 bytes and consists of 85 different values. In the following sections, the structure itself is described in-depth and divided into 3 listings. The first part of the data structure can be seen in Listing 1. All important variables are described in the following sections and commented on in the actual code shown in the listings.

The first part of the packet contains a race status indicator with a timestamp. Maximum engine rpm and idle rpm values together limit the rpm spectrum of the engine. The current engine rpm value lies in the mentioned range and indicates the actual revolutions per minute of the engine. The rest of the data in the first part of the packet is mostly used as input to motion racing rigs, and will not be utilized by the Racing Dashboard application. Although they will not be used, a summary may come in handy. Acceleration and velocity are present in all three axes, as well as angular velocity. The yaw, pitch, and roll of the vehicle, the suspension travel of each wheel, the slippage of the tires, and the speed of the wheels are also available. All data could be

```
struct ForzaHorizon5DataPacket {
    let IsRaceOn: Int32 // 0 - Off, else - On
    let TimestampMS: UInt32 // Timestamp
    let EngineMaxRpm: Float32 // Max RPM of the engine
    let EngineIdleRpm: Float32 // RPMs of the iddling engine
    let CurrentEngineRpm: Float32 // Actual engine rpm
    let AccelerationX: Float32 = 0
    let AccelerationY: Float32 = 0
    let AccelerationZ: Float32 = 0
    let VelocityX: Float32 = 0
    let VelocityY: Float32 = 0
    let VelocityZ: Float32 = 0
    let AngularVelocityX: Float32 = 0
    let AngularVelocityY: Float32 = 0
    let AngularVelocityZ: Float32 = 0
    let Yaw: Float32 = 0
    let Pitch: Float32 = 0
    let Roll: Float32 = 0
    let NormalizedSuspensionTravelFrontLeft: Float32 = 0
    let NormalizedSuspensionTravelFrontRight: Float32 = 0
    let NormalizedSuspensionTravelRearLeft: Float32 = 0
    let NormalizedSuspensionTravelRearRight: Float32 = 0
    let TireSlipRatioFrontLeft: Float32 // 0 - grip,
    // ratio > 1 loss of grip
    let TireSlipRatioFrontRight: Float32 = 0
    let TireSlipRatioRearLeft: Float32 = 0
    let TireSlipRatioRearRight: Float32 = 0
    let WheelRotationSpeedFrontLeft: Float32 // radians/sec.
    let WheelRotationSpeedFrontRight: Float32 = 0
    let WheelRotationSpeedRearLeft: Float32 = 0
    let WheelRotationSpeedRearRight: Float32 = 0
    // ... More data
}
```

Listing 1: First part of the FH5 data structure

used by force feedback accessories to exhibit events such as car roll, slippage, or collisions.

The second part of the Forza Horizon 5 data structure encapsulates rumble and puddle detection for each wheel independently. These properties have the prefix *WheelInPuddle* and *WheelInRumble*. In the case of *WheelInPuddle*, the value ranges from 0 to 1, where 1 is the deepest puddle and 0 is the shallowest. On the other hand, *WhellInRumble* is simpler and can only

```
1  struct ForzaHorizon5DataPacket {
2      let WheelOnRumbleStripFrontLeft: Float32 = 0
3      let WheelOnRumbleStripFrontRight: Float32 = 0
4      let WheelOnRumbleStripRearLeft: Float32 = 0
5      let WheelOnRumbleStripRearRight: Float32 = 0
6      let WheelInPuddleDepthFrontLeft: Float32 = 0
7      let WheelInPuddleDepthFrontRight: Float32 = 0
8      let WheelInPuddleDepthRearLeft: Float32 = 0
9      let WheelInPuddleDepthRearRight: Float32 = 0
10     let SurfaceRumbleFrontLeft: Float32 = 0
11     let SurfaceRumbleFrontRight: Float32 = 0
12     let SurfaceRumbleRearLeft: Float32 = 0
13     let SurfaceRumbleRearRight: Float32 = 0
14     let TireSlipAngleFrontLeft: Float32 = 0
15     let TireSlipAngleFrontRight: Float32 = 0
16     let TireSlipAngleRearLeft: Float32 = 0
17     let TireSlipAngleRearRight: Float32 = 0
18     let TireCombinedSlipFrontLeft: Float32 = 0
19     let TireCombinedSlipFrontRight: Float32 = 0
20     let TireCombinedSlipRearLeft: Float32 = 0
21     let TireCombinedSlipRearRight: Float32 = 0
22     let SuspensionTravelMetersFrontLeft: Float32 = 0
23     let SuspensionTravelMetersFrontRight: Float32 = 0
24     let SuspensionTravelMetersRearLeft: Float32 = 0
25     let SuspensionTravelMetersRearRight: Float32 = 0
26     let CarOrdinal: Int32 = 0
27     let CarClass: Int32 = 0
28     let CarPerformanceIndex: Int32 = 0
29     let DrivetrainType: Int32 = 0
30     let NumCylinders: Int32 = 0
31     // ... More data
32 }
```

Listing 2: Second part of the FH5 data structure

be 1 or 0, where 1 indicates that the wheel is on a rumble strip and 0 means that the wheel is on solid ground. The values prefixed by *SurfaceRumble* are non-dimensional and used for force feedback. Tire slip angles and combined tire slip values are not well documented. The suspension travel is measured in meters for each wheel. All the variables mentioned above are often common to other racing simulators. In contrast, *CarOrdinal*, *CarClass*, and *CarPerformanceIndex* are specific for Forza Horizon 5, where *CarOrdinal* is the unique ID of the car, *CarClass* rates exclusivity, ranging from 0 (worst)

to 7 (best) and *CarPerformanceIndex* starts with the value 100 and ends with 999. The higher the value, the faster the car will run. *DrivetrainType* can be 0 (front-wheel drive), 1 (rear-wheel drive), and 2 (all-wheel drive). The last value in the described section is *NumCylinders*, that is, simply the number of engine pistons. The structure of the second section of the data packet can be seen in Listing 2.

```
1   struct ForzaHorizon5DataPacket {
2       let PositionX: Float32 = 0
3       let PositionY: Float32 = 0
4       let PositionZ: Float32 = 0
5       let Speed: Float32 = 0
6       let Power: Float32 = 0
7       let Torque: Float32 = 0
8       let TireTempFrontLeft: Float32 = 0
9       let TireTempFrontRight: Float32 = 0
10      let TireTempRearLeft: Float32 = 0
11      let TireTempRearRight: Float32 = 0
12      let Boost: Float32 = 0
13      let Fuel: Float32 = 0
14      let DistanceTraveled: Float32 = 0
15      let BestLap: Float32 = 0
16      let LastLap: Float32 = 0
17      let CurrentLap: Float32 = 0
18      let CurrentRaceTime: Float32 = 0
19      let LapNumber: UInt16 = 0
20      let RacePosition: UInt8 = 0
21      let Accel: UInt8 = 0
22      let Brake: UInt8 = 0
23      let Clutch: UInt8 = 0
24      let HandBrake: UInt8 = 0
25      let Gear: UInt8 = 0
26      let Steer: Int8 = 0
27      let NomalizedDrivingLine: Int8 = 0
28      let NormalizedAIBrakeDifference: Int8 = 0
```

Listing 3: Third part of the FH5 data structure

The third and at the same time the last Forza Horizon 5 data packet part contains information about the vehicle's position on the map, defined by its coordinates in the x, y, and z axes. The following mentioned values play a significant role in dashboard applications, and most of them will be used within the Racing Dashboard application. Among these values is the vehicle's speed in meters per second, followed by torque and speed, where both define the en-

gine's current output. In addition, the temperature of each tire, the pressure of the boost system, and the fuel level are included. Among the values that are often monitored is the usage of the brake, gas, and clutch pedals, as well as the gear-in-use indicator. For the race itself, data for the best lap time, last lap time, current time, race position, and current lap are also included in the packet. The third part of the described packet can be seen in Listing 3.

Chapter 3

Used Technologies

Based on the project requirements, technologies and frameworks suitable for this particular application were chosen. Although the platform was determined by the assignment, the programming language was not. There are several options to choose from; however, Swift has been chosen because it is the primary language for iOS development. Several third-party frameworks were selected, such as the UI constraints library SnapKit or Charts, which is used to create various graphs. Apple's Xcode is used as the main IDE for the entire development process. The networking layer of the application is powered by the SwiftNIO library. All technologies used are described in detail in the following sections.

3.1 GUI Creation Methods in iOS

In terms of iOS development, the GUI can be created using multiple methods within the Xcode IDE. Each method offers different tools to work with. The methods are as follows:

- **Storyboards:** Storyboard is a visual tool for creating multiple views, transitions between them, and arranging the elements within. As the name implies, the Xcode Storyboard acts as a board for building the GUI. The design process is intuitive and fast, although it has some disadvantages. The first problem is version control and inevitable merge conflicts, due to the machine-generated representation of the storyboard, which is nearly unreadable by a human. Furthermore, large storyboards are difficult to navigate and maintain [20].
- **NIBs/XIBs:** NIB/XIB files describe the user interface and are generated using the Interface Builder. XIBs are files used in the development process, whereas NIBs are generated upon build. NIBs/XIBs are the predecessors of storyboards and share the same version control issues [20].

- **Programmatic GUI:** GUI created using a programmatic approach is defined solely by code. This lower-level method has virtually no restrictions in terms of possibilities. There are no version control issues, such as complicated merge conflicts. The biggest disadvantage is a slower GUI development process, making this approach less suitable for prototyping.
- **SwiftUI:** The newest addition to the iOS GUI creation methods is Apple's SwiftUI, which defines the user interface declaratively and offers rapid application building. SwiftUI features well-known elements, such as lists, stacks, buttons, and more. Although the syntax and usage are quite different from the programmatic approach, internally, SwiftUI is still using frameworks such as UIKit. The biggest advantage of SwiftUI is fast prototyping and creating GUIs with fewer lines of code. Although the framework was introduced in 2019, fast-paced development made SwiftUI a capable tool for creating GUI [21].

Based on the requirements of this project, the programmatic approach was chosen. NIBs/XIBs were omitted because they are considered outdated. Storyboards do not offer high customization and are not preferred from a subjective point of view. SwiftUI could be used; however, it still offers weaker customization than the programmatic approach. To satisfy the requirements of the application, high customization is fundamental.

3.2 Swift

Swift is an open source, multi-paradigm, compiled and high-performance system programming language that is used primarily for the development of applications for the iOS, macOS, watchOS, and tvOS platforms. It was intended as a successor to well-established Objective-C. As a result of being compiled using the LLVM compiler, Swift allows running Objective-C, C, and C++ codes within one program. Despite the usage of LLVM, Swift is not a C-derived language. Swift is categorized as an object-oriented, strongly typed programming language with the support of imperative, declarative, and functional programming. Swift does not utilize a garbage collector for memory management; instead, it uses automatic reference counting (ARC). Thus, the user is partially involved in memory management, and memory leaks can occur in some cases. Swift is the only programming language used in this project.

3.3 Xcode IDE

Xcode is Apple's IDE used to develop software for the iOS, macOS, iPadOS, watchOS, and tvOS platforms. Xcode supports the creation, testing, and submission of applications. It comes free with the macOS operating system and is exclusively for macOS; therefore, it cannot be used with Windows or

Linux. It supports a variety of languages, including Swift, Objective-C, C, C++, Java, Python, and more. For debugging purposes, Xcode offers an iOS simulator that can replicate real iOS devices, in addition to other platforms. Furthermore, Xcode features a wide range of functionality, including a built-in Interface Builder that can be used for GUI design without the need for code. The Interface Builder will not be used in this project, due to versioning issues and overall poorer scalability [22].

3.4 Cocoapods

CocoaPods is a dependency manager for Swift and Objective-C Cocoa projects. It contains almost 89 thousand libraries [23] and provides a standard format for managing dependencies. CocoaPods focuses on a source-based distribution of libraries with an automatic way of integration into Xcode, maintaining versions and dependencies between libraries. The manager runs on the command line and is easy to install. After the initialization of CocoaPods, a Podfile is created, which contains the list of libraries that are being used. Libraries can be added or removed by alternating the Podfile. The software resolves any additions or changes to the dependency list. CocoaPods was chosen for this project because of its straightforward usage and reliability. Additionally, all third-party libraries used within this application are installed and managed via CocoaPods, with the exception of SwiftNIO, that is managed by Swift Package Manager. [23].

3.5 Swift Package Manager

The Swift Package Manager is a dependency manager that is used to distribute code. As an integrated part of the Swift build system, it manages the processes of downloading, compiling, and linking dependencies [24]. Xcode 11 and up integrates the package manager with support for iOS, macOS, watchOS, and tvOS application packages. [25]. Dependencies are modules that are required by the code in the package, whereas a package consists of Swift source files and a manifest file. The Swift Package Manager reduces coordination costs by automating the process of downloading and building all dependencies for a given project. It is an Apple's official alternative to other package managers such as CocoaPods or Carthage.

3.6 Frameworks

Various frameworks were chosen directly for this project; some of them are Apple's native frameworks, and others are third-party frameworks. Without the frameworks, this project would be much more difficult to complete.

The essential frameworks used within the racing dashboard application are described in the following subsections.

3.6.1 UIKit

Apple's UIKit framework allows one to build and manage a graphical user interface based on events for iOS and tvOS applications. UIKit provides a window and view architecture, which is the core of GUI development. Additionally, the event handling infrastructure allows for capturing input events such as Multi-Touch. The framework offers animation support, document support, drawing, search support, and resource management, among others [26].

3.6.2 SnapKit

SnapKit is a third-party framework that offers a domain-specific language that is used to make the Auto Layout much easier to use. Both the iOS and macOS platforms are supported. In other words, SnapKit allows one to create GUI constraints with minimal effort, such as positioning, sizing, setting margins, and more. Expressive chaining of the language provides great readability. The framework is type-safe; therefore, it reduces errors caused by programmers. SnapKit can be installed through dependency managers such as CocoaPods, Carthage, and Swift Package Manager [27].

3.6.3 Swinject

Swinject is a lightweight dependency injection framework designed for Swift. It is powered by a generic type system and first-class functions to resolve dependencies [28]. Swinject's workflow is based on a Container type that is used to register dependencies. Once a dependency is registered, it can be resolved and, therefore, retrieved from the container. Furthermore, the framework features assemblers, assemblies, storyboard support, and can be expanded to provide an auto-register feature. Swinject can be installed through dependency managers such as CocoaPods, Carthage, and Swift Package Manager.

3.6.4 Combine

Combine is Apple's reactive framework similar to ReactiveSwift or RxSwift. It is used to handle asynchronous events over time or, in other words, to process values over time. The Combine framework provides a declarative Swift API based on publishers and subscribers. Publishers can deliver a sequence of values over time. On the other end of the communication, a subscriber stands. Subscribers act on the elements as they receive them. Multiple publishers can be combined using different operators, resulting in a new publisher. Some of the publisher operators are map, filter, collect, zip, and many others. One way

to think about reactive programming is to imagine a stream of data. Data are generated on one side, flowing through the stream up to the other side, where a subscriber awaits any changes and reacts accordingly [29].

3.6.5 SwiftNIO

Considerable attention was paid when deciding which networking framework to use. In total, three frameworks were taken into account, namely CocoaAsyncSocket, Apple Network, and SwiftNIO. CocoaAsyncSocket is a famous third-party networking library for Swift, which supports both TCP and UDP protocols. In 2018, Apple introduced at WWDC its own version named Apple Network[30], which is a native networking framework that aims to replace other third-party frameworks, such as CocoaAsyncSocket. The Network framework supports protocols such as TLS, TCP, or UDP. Being a native framework developed by Apple is a considerable advantage over other frameworks. The last framework considered is SwiftNIO [31], which is a server-side, asynchronous, event-driven, and non-blocking Swift framework used to build network applications. SwiftNIO features a straightforward setup that makes it easy to integrate into a project. The combination of a high-performance underlying layer and a simple setup makes SwiftNIO the best candidate for this project. The framework is presented to be maintainable and suitable for rapid development. SwiftNIO shares similarities with Netty [32], with the difference of being written for Swift. The framework implements the following low-level protocols: HTTP/1, HTTP/2, WebSocket, TLS, and SSH. The supported high-level protocols are as follows: HTTP, gRPC, APNS, PostgreSQL, and Redis [31]. Besides other features, SwiftNIO's UDP capabilities, such as the UDP server, are suitable for this project.

3.6.6 CoreData

CoreData is Apple's approach to a object graph management and persistence framework for macOS and iOS SDKs [33]. It allows developers to create high-performance data-driven applications. CoreData can store and retrieve data; however, it is not a relational database like MySQL; instead, CoreData is more of a collection of objects that are in relationship. Additionally, the framework takes care of the life cycle of objects in the graph [34]. CoreData is generally used to persistently save data from an application, cache temporary data, and add undo functionality [35]. The framework features a Data Model Editor that lets the developer define data types, relationships, and generate respective class definitions that can be used further within the project.

3.6.7 Charts

Charts is a third-party library that provides various types of graphs that can be used to display different types of data. The library originated as a port of

the well-known Android library MPAndroidChart [36]. Charts enable the developer to create fully-functional charts using a few lines of code. The core features of the library are eight types of charts with various additions such as axis scaling, dragging, panning, a combination of charts, fully customizable axes, legends, and many more. Charts can be installed using managers such as CocoaPods, Carthage, or Swift Package Manager [37].

3.6.8 XCTest

XCTest is Apple's framework for creating various types of tests, such as unit tests, performance tests, and UI tests. The tests are seamlessly integrated with the testing workflow of Xcode. The framework features various asserts that are used to check whether certain conditions are satisfied or not. The common asserts are XCTAssertEqual, XCTAssertNil, XCTAssertTrue, XCTAssertFalse and many others [38].

3.6.9 SwiftLint

SwiftLint is a tool for enforcing style and conventions for Swift. Like all other linter tools, SwiftLint enforces a certain code style, in this case, a set of common rules generally accepted by the Swift community [39]. Using SwiftLint helps to maintain code quality.

Chapter 4

Design

4.1 Architecture

The application architecture has a direct impact on the development process itself. Incorrectly chosen architecture will negatively affect the quality of the implementation and any further expansion of the application. In terms of iOS development, two common architectures are used: MVC and MVVM. Apple's recommendation is to use MVC, which stands for Model-View-Controller. MVC splits the program logic into three elements: data, logic, and view. Although the pattern is well-known and well-established, it often leads to a complication called a massive view controller. In the case of iOS development, the massive view controller issue occurs frequently and can be solved for the most part by using the Model-View-ViewModel architecture, MVVM shortly.

4.1.1 MVVM

Model-View-ViewModel, or shortly MVVM, is a structural and architectural pattern used for the separation of the application's logic into three layers. MVVM offers great maintainability, extensibility, and testability. The mentioned layers of MVVM are as follows:

- **Model** encapsulates data, manages the business logic of the application, and the state of the application. Commonly represented by structs or classes.

- **View** presents visual elements on the screen, such as buttons, labels, pictures, videos, etc. In addition, the view handles user input.

- **View model** handles the transformation of model data into displayable values. Serves as a link between the model and view. The view model

does not have a reference to the view. The view model usually updates the view using data binding.

Figure 4.1 shows a diagram of the MVVM architecture, which consists of three main layers and the relations between them. Dashed arrows represent non-reference binding and, conversely, full arrows represent references.

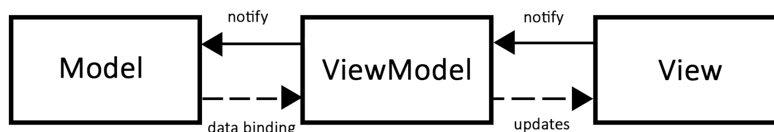

Figure 4.1: Model-View-ViewModel diagram [40]

4.1.2 MVVM in iOS

Adaptation of the MVVM pattern in iOS must incorporate view controllers, which are a crucial building block of iOS applications. The modified diagram of the MVVM that is being used in iOS is shown in Figure 4.2.

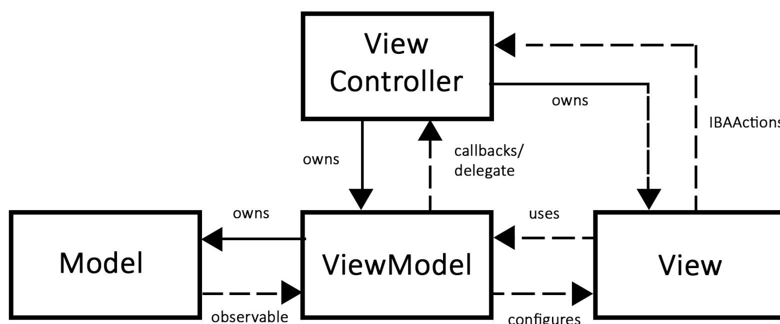

Figure 4.2: Model-View-ViewModel in iOS diagram [40]

4.1.3 Flow Coordinators

Managing the transitions between screens in large-scale projects can become overwhelming very quickly. Without coordinators, view controllers are responsible for navigation and; therefore, less reusable. A coordinator is a plain object that takes the navigation responsibility from the view controller and handles it by itself. This makes the view controller more manageable and reusable. Navigation logic is not scattered around, but grouped into one logical layer. Every coordinator can feature multiple child coordinators creating an n-ary tree structure. This approach allows the coordination process to be

split into multiple subprocesses, expanding the abstraction even further, thus making it more manageable. The view controller sends messages to its respective coordinator, who handles them accordingly. The coordinator decides whether the message leads to events, such as loading a new view controller or dismissing one. Furthermore, the view controllers are not aware of other view controllers. They only communicate with the responsible coordinator [41].

4.2 Mobile First Design

Mobile-first design is a method of designing the user interface by focusing on mobile devices first, prioritizing the smallest screen and working its way up to larger screens, such as tablets or computers. This approach is often used for website development. The strategy is based on the fact that smaller screens will fit fewer content, and the UX designer has to prioritize key aspects of the product. This forces the designer to pinpoint the most important UX components. Once a small-screen mobile design is completed, larger screens can be derived with the addition of some other detailed elements [42]. In terms of iOS development, the mobile-first design approach is not as mandatory as in web development, where screen sizes differ much more. However, the size difference between the Apple Watch, iPhone, and iPad devices is significant enough to utilize the mobile-first design approach. The application developed within this project will support iPhone and iPad devices; therefore, the mobile-first design can be applied.

4.3 Separation of Concerns

Separation of concerns (SoC) is a fundamental principle in software engineering. The concept is based on the separation of the program into multiple distinct sections where all sections have a specific purpose. Each section addresses a single concern without having any knowledge of the rest of the sections. The program is essentially a combination of all sections. The SoC enables good testability as a result of its high modularity. Each section can be tested independently.

4.4 Dependency Injection

Dependency injection (DI) is a fundamental and easy-to-adopt design pattern that implements Inversion of Control. Application of the pattern makes the code loosely coupled; therefore, easier to manage and test. In the case of not using dependency injection, an object creates its dependencies by itself. This approach makes the object responsible for dependency management and creates coupling. Dependency injection pattern moves the dependency creation process outside the object, and all dependencies are injected into

it. In other words, dependency injection means giving an object its instance variables [43].

4.5 Localization

Localization is the process of adapting an application to various languages and regional specificities. Localized applications report a higher success rate in respective markets than applications that do not support the given market [44]. Values such as strings, dates, regional numeric formats, units of measurement, and time are typical candidates for localization. The selected localization is usually derived from the region and also from the language of the given country. Each application should support localization, even if it initially supports only one language [44].

4.6 Graphical User Interface

The design of the graphical user interface is a key step during the application development process. As discovered in the analysis section of this thesis, most existing racing dashboard applications suffer from an unsatisfactory GUI. The application created within this project takes advantage of this deficiency and focuses on a modern and intuitive user interface. Each designed screen is described in detail in the following sections. During the design of the interface, the low-fidelity wireframe phase was omitted. Instead, high-fidelity prototypes were immediately created. This decision may seem unconventional; however, it is based on several arguments. As mentioned above, the visual part of the application is crucial. To satisfy this requirement, the design process must be tightly coupled with the respective functionality of the UI elements. During the design, the high-fidelity prototype offered an immediate preview of the application. In this way, the design or even functionality could be slightly adapted to create a balance between intuitive controls and graphical design. Although the UIKit framework 3.6.1 offers many iOS elements and features, most of the content used within the application was highly customized. In the following sections, all designed screens are described in detail, including the respective high-fidelity prototypes. The featured wireframes are only for iPhone devices. Due to the mobile-first approach, the design and layout for iPads remain nearly the same with a few exceptions. For this reason, it was not necessary to also create wireframes for the iPad.

4.6.1 Main Menu Screen

The main screen of the application is the main menu. The screen is primarily used to start the connection process with a selected combination of a simulator and a dashboard style. Furthermore, the screen offers an overview of the last

race statistics and a start button. A high-fidelity prototype of the main menu screen is shown in Figure 4.3. The key elements of the screen are the following:

- **Application name and version labels:** In the upper left corner are two labels. The bold label indicates the application name, and the smaller label displays the version of the release.
- **Simulator/game selector:** The simulator selector is located on the left side of the screen. It is a custom built selection element for browsing and selecting simulators using a swipe gesture. The selection is accompanied by a haptic feedback, and the direction of swiping is indicated by moving chevrons.
- **Dashboard style selector:** Dashboard style selector shares nearly all properties with the simulator with the difference of selecting dashboard styles instead of simulators.
- **Last record statistics:** The right portion of the screen is dedicated to statistical data from the last race. The section displays the following information: race date, simulator name, best lap, total time, top speed, and final position. Values in titles are accompanied by respective icons. Finally, there is an orange graph button next to the simulator name that moves to the graph screen when activated.
- **Start button:** Located within the statistics section on the bottom right side, the start button segues to a connection screen with a given simulator and dashboard style when activated.

4.6.2 Graphs Screen

The graph screen offers a statistical overview of the races recorded. The right portion of the screen is almost identical to the main menu's one, except the start button, which is replaced by two buttons used for switching records. Furthermore, every record offers additional data displayed using a graph. The data include the usage of the accelerator, brake, and gear during the race. A high-fidelity prototype of the graph screen is shown in Figure 4.4. The key elements of the screen are the following:

- **Last record statistics:** The right part of the screen dedicated to statistical values is nearly the same as the one mentioned in the main menu screen. The key difference are the buttons on the bottom. The next and previous buttons are introduced instead of the start button. Both buttons serve as controls for record switching. The button is disabled when no further record is available. The disabled state is indicated by a color change.

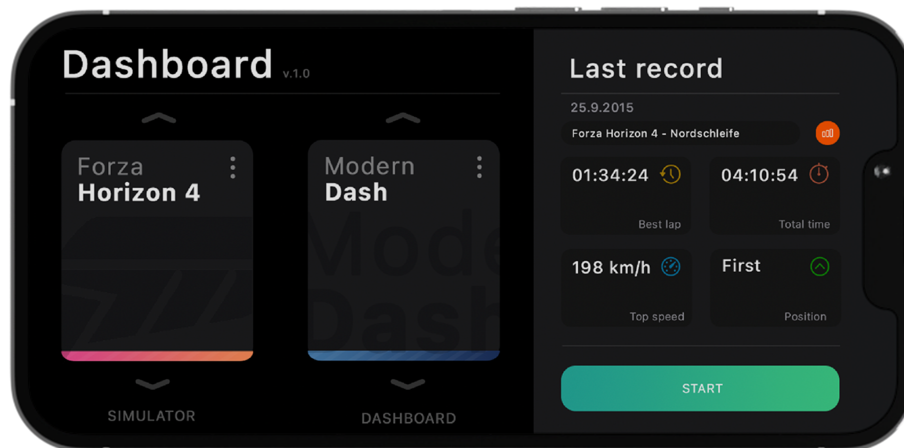

Figure 4.3: High-fidelity prototype of main menu

- **Data type label:** The data type label located above the graph displays the name of the selected data type that is being displayed in the graph.
- **Back button:** In the top left corner, a back button is located. The button navigates back to the main menu when activated.
- **Graph:** Majority of the left part of the screen is taken up by the graph that displays different values in relation to race time. The values shown can represent either the accelerator pedal, the brake pedal, or the gear usage. Every data type is drawn with a different color. The graph offers scaling on the x-axis by pinching. The y-axis displays the given values, and the x-axis represents the race time. The scale of the y-axis is in case of the accelerator and brake pedals between 0 and 100, whereas the range for the gear is dynamic and depends on the received values. The graph is created using the Charts library 3.6.7.
- **Data selector:** The data type selector is attached to the bottom boundary of the graph. Its purpose is to select the data displayed by the graph. The gas pedal option is implicitly selected.

4.6.3 Connection Screen

The screen used for the connection process is divided into two sections just like the main menu or the graph screen, even sharing the same division ratio, thus making the GUI more homogeneous. The screen guides the user through

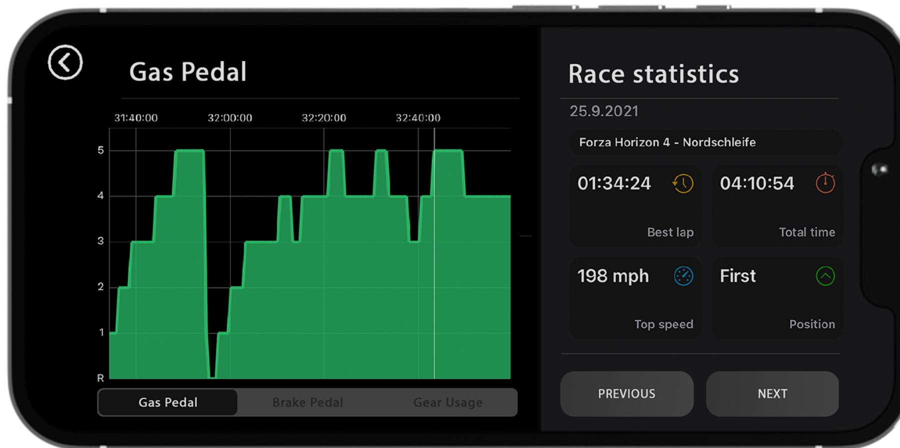

Figure 4.4: High-fidelity prototype of graphs screen

the connection process, showing the status and IP address with the port when successfully connected. Furthermore, a simple swipe gallery shows the step by step connection process for the chosen simulator. Once connected, the user is allowed to launch the dashboard. A high-fidelity prototype of the connection screen is shown in Figure 4.5. The key elements of the screen are the following:

- **Swipe gallery:** Every simulator has a specific way to configure the data output feature. To avoid any difficulties with the setup, a swipe gallery with the respective instructions is shown, enabling the user to scroll through the guide. An adjacent slide indicator shows which slide is currently being displayed. One slide consists of an instruction label accompanied by an in-game setup screenshot.

- **Connection label:** The connection label located in the right section is a static header.

- **Connection status:** The connection status indicator consists of an animated icon and a label that displays the connection state. Once the device is connected, the icon becomes a checkmark, and the message is updated accordingly.

- **IP address panel:** The IP address panel displays the IP address of the UDP server accompanied by an icon with a description label.

- **Port panel:** The port panel displays the port of the UDP server accompanied by an icon with a description label.

- **Continue button:** Once connected, the continue button is enabled and navigates to a dashboard when activated.
- **Close button:** In the top right corner, a close button is located, which is used to abort the connection process by returning to the main menu.

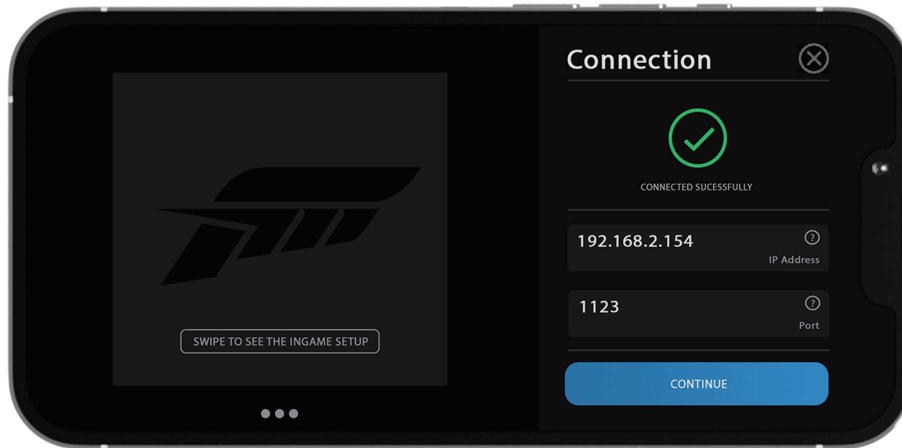

Figure 4.5: High-fidelity prototype of the connection screen

4.6.4 Dashboard Screen

The dashboard included within the release of the application is based on a modern flat design. It displays fundamental data about the race and vehicle. A modern color scheme is used to increase the contrast of each element leading to better clarity.

- **Position indicator:** The top left corner is occupied by a position indicator that informs the player about the position of the driver within the race.
- **Lap indicator:** The element at the top right corner is styled the same as the position indicator, except it displays the current lap number.
- **Clutch, brake and gas pedal usage:** Vertical gauges located on the left side of the screen are used to display the usage of clutch, brake, and gas pedals. Each vertical gauge allows visualization of any value ranging from 0 to 100. In addition, every gauge has an adjacent description label, gray background, and a configurable colored progress bar.

- **RPM gauge:** Engine RPMs of the vehicle are shown using a 270 degree circular gauge. The color of the strip gradually changes from blue to red. Red indicates high rpm; on the contrary, blue indicates low rpm.
- **Speed indicator:** The speed indicator is a label that shows the current speed of the vehicle, including a measurement unit label.
- **Gear indicator:** The gear indicator is styled the same way as the speed label, except that it displays the gear in use. In addition, the background of the label is colored red when the RPMs of the engine reach a specific value, acting as a shift light.
- **Tire temperature:** The simplified car layout shows the temperature of each tire independently. The temperature is displayed in red color with the opacity gradually changing in relation to the temperature of the given tire.
- **Fuel level:** The fuel level gauge shares some similarities with the pedal gauge, except that the gauge is horizontal with a fuel icon and letters indicating empty or full endpoints.
- **Best lap time:** The best lap time label is formatted as mm:ss:ms. It displays the race time of the best lap so far. A description label is included.
- **Total time:** The total time is styled as a horizontally mirrored best lap indicator. It displays the total race time in the same format, mm:ss:ms.

4.6.5 Application Icon

The application icon is an integral part of an application due to its representative aspect. The design process was strongly affected by Apple's guidelines [1], reflecting terms such as simplicity, single focus point, simple background, and more. The idea behind the designed icon is to represent a simple circular gauge that is commonly used in cars. The gauge shows a static value indicated by the needle and the circular strip. Tick marks are included for a more authentic appearance. Furthermore, a graph sketch is included at the bottom of the icon to indicate the application's support of racing statistics. The final icon design can be seen in Figure 4.7.

Figure 4.6: High-fidelity prototype of modern dashboard

Figure 4.7: Application icon

4.7 Race Statistics Database

As stated in the assignment, data sets are recorded for every session regardless of whether a race is in progress or not. Once the dashboard is loaded, the data recording begins fully automatically. To persistently save any kind of data, a database is needed; therefore, a database structure design process is mandatory. The chapter concerned with the used technologies covered Core-Data as the chosen persistence framework. As far as the designing process goes, the conceptual entity relation diagram must be designed first. It serves as a blueprint for the database construction process. The diagram is shown in Figure 4.8.

The race record entity is created as the result of a race session. Each record contains data such as finishing position, time of the best lap, top vehicle's speed, total time of the race or session in case of absence of a race, name of the race track, racing simulator name, and date of the record. In

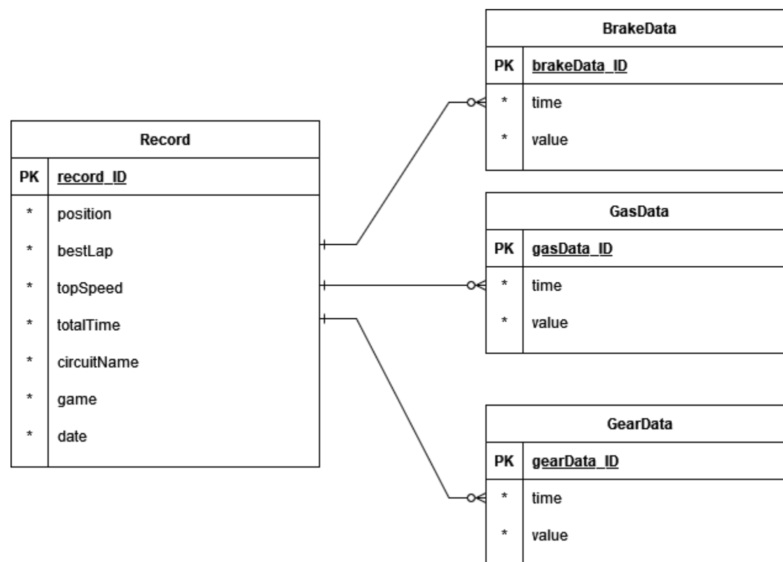

Figure 4.8: Conceptual diagram of race records database

addition, the usage of three attributes, namely the use of gas, brake, and gear is recorded. These values are represented by respective entities related to the record entity. It is evident from the diagram 4.8, that every record is able to optionally store multiple entries for each type of data set. Implementation with CoreData is covered in the upcoming chapter.

4.8 Common Data Packet

Part of the analysis process was to describe a data packet from an arbitrary racing simulator. The chosen data packet was from Forza Horizon 5 and is described in section 2.5. Data packets from other racing simulators differ in structure and data provided. A good architectural approach is to implement a data structure to which all other packets can be converted. This solution helps to avoid any problems when using different packets. During the design of the common data structure, the data intersection of the Forza Horzion 5 and Assetto Corsa packets was used to determine the common values for both simulators. Other simulators are assumed to have the same common values in case they are included in the future.

The common data structure can be seen in Listing 4, and it is easy to notice that the structure is similar to the Forza Horizon 5 data packet. The application will support all listed values in the data structure, although this does not necessarily mean that all dashboard designs will utilize all the data. The data itself are not only copied from the original packet to the common data structure but are also transformed and normalized for the application's use. For ex-

ample, the accelerator pedal value in Forza Horizon 5 ranges from 0 to 255, which can be represented by a single byte. On the other hand, the common data packet represents the value of the accelerator pedal as a fractional number ranging between 0 and 1. Furthermore, the common data packet stores measurement values in a metric system, and conversion to the desired measurement system occurs within the logical section of the application, where the data preparation for rendering happens.

4.9 Networking Layer

The networking functionality is the basis for the entire application; therefore, the design step is crucial. As a result of the analysis, the UDP protocol was determined to be commonly used in racing simulators to pass data. The communication flow consists of a server on one side and optionally multiple clients on the other. In terms of this project, the Racing Dashboard application acts as a UDP server and, conversely, the racing simulator acts as a client. The server needs to be hosted using the IP address of the device running the application and a port. The port can be set to any port number that is not used by another application. The host address is the same as the assigned IP address of the device in the local network.

Once the UDP server is hosted, the IP address and port are used in the given racing simulator data output settings. After providing the simulator with the IP address and port, it starts to act as a client by sending data to the server. The networking layer receives all the input data and provides them for transformation and rendering. SwiftNIO framework features the ability to create a custom UDP server that can be used by the Racing Dashboard application.

```
struct UIDataPacket {
    let isRaceOn: Int32?
    let timestampMS: UInt32?
    let engineMaxRpm: Float32?
    // Current engine rpm (revs per minute)
    let currentEngineRpm: Float32?
    // Vehicle speed in m/s
    let speed: Float32?
    let power: Float32?
    let torque: Float32?
    // Tire temperature in degrees celsius
    let tireTempFrontLeft: Float32?
    let tireTempFrontRight: Float32?
    let tireTempRearLeft: Float32?
    let tireTempRearRight: Float32?
    let boost: Float32?
    // Fuel level ranging from 0 to 1
    let fuel: Float32?
    // Time of the best lap
    let bestLap: Float32?
    let lastLap: Float32?
    // Time of the current lap
    let currentLap: Float32?
    // Time in seconds with milisec fraction: 10.287 - 10s, 287ms
    let currentRaceTime: Float32?
    // Current lap, starting from 1
    let lapNumber: UInt16?
    // Position in the given race
    let racePosition: UInt8?
    // Accel pedal intesity ranging from 0 to 1
    let accel: Float32?
    // Braking intesity ranging from 0 to 1
    let brake: Float32?
    // Clutch intesity ranging from 0 to 1
    let clutch: Float32?
    let handBrake: UInt8?
    // 0 - Reverse, 1,2,3.. Gears
    let gear: UInt8?
    let circuitName: String?
}
```

Listing 4: Common data structure

Chapter 5

Implementation

The implementation chapter focuses on the implementation of the Racing Dashboard application. Several parts of the implementation are due to their importance described in more detail than others. The chapter begins with the most important part of the application, the networking layer. The networking layer section covers the networking service itself, including channel handlers and decoders. The networking layer encapsulates the service and several important parts of the core application logic are described, such as the connection process or data recording functionality. Furthermore, the implementation of the chosen architecture is outlined, as well as the implementation of various GUI elements.

Listing snippets are a convenient way to ensure that the implementation is understandable and easy to comprehend. In order to avoid unnecessarily long code figures, only the fundamental parts of the implementation code are featured and adequately commented on.

5.0.1 Networking Service

The first step is to define the `NetworkServicing` protocol to which the concrete implementation will conform. The protocol is shown in Listing 5. This protocol-based approach ensures good testability and is commonly used with the dependency injection technique. The `NetworkServicing` protocol consists of a Combine stream that publishes the incoming raw data represented as a `ByteBuffer` with no failure option. Aside from that, `startListening(...)` and `stopListening()` methods are used to control the state of the UDP server. It is worth mentioning that the method used for starting the server can throw an error and must be handled appropriately.

The concrete implementation of the `NetworkServicing` protocol is called `NetworkService`. This class implements bodies of both mentioned functions for the server controls. Furthermore, the implementation introduces a `Channel` property used as the main object for communication. As can be

```
1 protocol NetworkingServicing {
2     // Stream of raw (bytes) data from the network
3     var rawData: PassthroughSubject<ByteBuffer, Never> { get }
4     // Stop listening to incomming data
5     func stopListening()
6     // Start listening to incomming data
7     func startListening(host: String?, port: Int) throws
8 }
```

Listing 5: Networking service protocol

seen in Listing 6, the `stopListening()` function has a straightforward implementation that only closes the communication channel. The slightly more advanced `startListening(...)` function implementation is shown in Listing 7. It takes a host IP address and a port as parameters. The code is separated by comments into three parts. First, a `MultiThreadedEventLoopGroup()` is created. This group contains `EventLoop` instances each bound to a thread. The `EventLoop` is used to process I/O tasks in an endless loop for a channel.

In the second part of the code, the group is bootstrapped to a `DatagramChannel` with a custom handler passed in the initializer. The option `SO_REUSEADDR` ensures that the port is reused even if it is currently busy (in `TIME_WAIT` state). It is set to the level of `SOL_SOCKET` with value of 1. Finally, in the last step, the bootstrap is bound to a host, using an IP address and a port. This operation can fail and return a `nil`, hence the `try?` keyword. Any thrown errors must be handled.

```
1 func stopListening() {
2     _ = channel?.close()
3 }
```

Listing 6: Networking service stop function

5.0.2 Channel Handler

The implementation of the channel handler is named `ChannelInboundHandler` and the most important part of the code is shown in Listing 8. It is, in fact, an object with a `channelRead(...)` method. This method receives inbound data as the function parameter. The data itself can be extracted using `unwrapInboundDataIn(...)` producing an addressed envelope that contains the raw data under its `data` attribute. The `passTroughSubject` is used to send data to the subscribers. It is important to note that the data in the cur-

```swift
func startListening(host: String, port: Int) throws {
    // ... Check host and port, handle errors
    // 1.
    let group = MultiThreadedEventLoopGroup(
        numberOfThreads: System.coreCount)
    // 2.
    let bootstrap = DatagramBootstrap(group: group)
        .channelOption(
        ChannelOptions.socket(
        SocketOptionLevel(SOL_SOCKET),
        SO_REUSEADDR),
        value: 1)
        .channelInitializer { channel in
            channel.pipeline
            .addHandler(
               EchoHandler(self.rawData))
        }
    // 3.
    channel = try? bootstrap.bind(host: host, port: port).wait()
    // ... Handle thrown errors
}
```

Listing 7: Networking service start function

rent state are represented as an array of bytes and require further operations, such as decoding, to work with it as with a normal object.

5.0.3 Data Decoder

In order the use the inbound data within the application for various operations and ultimately to be rendered by GUI, an encoding process must be integrated. Each supported racing simulator features a specific data format; therefore, a respective decoder must be implemented for each simulator separately. Each decoder must conform to a `DataDecoder` protocol, that defines a method used for decoding and is shown in Listing 9. The method takes a `ByteBuffer` as a parameter and returns a common data structure.

To explain the implementation of the decoder, the Forza Horizon 5 decoder class will serve as an example. The implementation itself is based on the extraction of specific bytes from the input data. Once the bytes are extracted using `getBytes(...)` method, they can be decoded into an object, in the case of Forza Horizon 5, to `ForzaHorizon5DataPacket`. It is crucial that the number of bytes extracted matches the size of the object to which it is parsing. Since the GUI uses a common data structure, a conversion method is

```
private final class EchoHandler: ChannelInboundHandler {

    typealias InboundIn = AddressedEnvelope<ByteBuffer>

    let passThroughSubject: PassthroughSubject<ByteBuffer, Never>

    public func channelRead(context: ChannelHandlerContext, data: NIOAny) {
        let addressedEnvelope = self.unwrapInboundIn(data)
        let envelopeData = addressedEnvelope.data

        // Send the packet to the passtrough subject
        passThroughSubject.send(envelopeData)
        // ...
    }
}
```

Listing 8: Networking service handler

part of every simulator-specific data packet, called `toUIDataPacket()`. This method simply uses data from the simulator-specific packet, transforms it to normalized values, and creates a common data structure encapsulation called `UIDataPacket`.

5.1 Application Logic

The logic behind the application is quite extensive and it is not necessary to explain it in its entirety. Parts of the implementation that are considered to be important or interesting are pinpointed and explained in this section.

5.1.1 Connection Process

The logic used to establish the connection between the application and a given racing simulator is embedded into the functionality of the connection screen. The process handles scenarios such as the repeated connection, error handling, and more.

The logic itself is encapsulated within the `ConnectionViewModel`. It features important methods for managing the connection, `startConnection()` and `stopConnection()`. Hosting a UDP server requires a port and an IP address. The fully custom networking helper class offers a `getWiFiAddress()` method, that checks the network interface of the device and provides its IPv4 address. In contrast, it returns `nil` if the address is not detected, which usually means that a WiFi connection is not established.

```
protocol DataDecoder {
    func decode(from data: ByteBuffer) -> UIDataPacket
}

final class ForzaHorizon5Decoder: DataDecoder {

    func decode(from data: ByteBuffer) -> UIDataPacket {
        var buff = ByteBuffer()
        // Racing simulator specific
        if let data = data.getBytes(at: 0, length: 232) {
            buff.writeBytes(data)
        }
        if let data = data.getBytes(at: 244, length: 79) {
            buff.writeBytes(data)
        }
        return buff.withVeryUnsafeBytes {
            $0.load(
            as: ForzaHorizon5DataPacket.self )
        }.toUIDataPacket()
    }
}
```

Listing 9: Networking service decoder

The method mentioned above for starting the connection instantiates a scheduled timer that invokes the network service connect function in a given time interval. The invoker tries to configure a connection continuously until it is successful. Once connected, the invoker stops. The connection itself uses methods from the networking service. While the connection invoker is running, the user receives a connection status, which can optionally consist of error messages. After finishing a race session and closing the dashboard, `stopConnection()` is automatically called. The function stops the listening service and invalidates the scheduled timer if it is still running. The reconnect interval is set to 1.5 seconds.

5.1.2 Data Recording

Each session of using the racing dashboard application produces a data record. The data source stream produced by the networking layer sends data over time represented as a common data structure object. This stream is mainly used to render the values on the screen, although the data recording functionality subscribes to it as well. Each time a data packet is received, all necessary values are updated within the local `RaceStatisticsRecord` object. In ad-

dition, the object features helper methods such as `updateTopSpeed(...)`, `addToGasDataSet(...)`, and more. The record object is updated using these methods for each new received data packet. However, there is one issue that must be addressed. The data stream can produce up to 60 packets per second. With this rate of updates, the records would reach an unnecessarily large size. To avoid this issue, Combine's `throttle(...)` operator was used. Its functionality is based on publishing only one element in the specified time interval. The chosen element from the interval can be either the most recent or the first, depending on the settings. An interval of 500 ms proved to be effective for data recording.

As mentioned before, during the session, the data are stored in a local variable. This approach would not be persistent, and the data would vanish after the application was closed. Therefore, after ending the session, method `saveRecord()` is called automatically. Once called, CoreData executes CoreData's save method on the current context located within the persistent container. This simple operation keeps the record persistent.

To retrieve the stored data, a fetch request is created using the `RaceStatisticsRecord.fetchRequest()`. After setting parameters such as the the fetch limit or the sorting descriptor, the request can be executed using `context.fetch(request)`. This method may fail and return a `nil`. Otherwise, it returns an array of data, in this case an array of records that are ready to be used. This fetch functionality is used within the graph screen that uses the data to render graphs and shows statistics, and also within the overview section of main menu.

5.2 Architecture

The application is based on the MVVM-C architecture, which stands for Model-View-VieModel-Coordinator. The chosen architecture is described in Chapter 4. Applying the MVVM means separating the project into 3 main building blocks. All view elements, including fundamental view controllers, belong to the view section. All business logic and data belong to the model, and in between stands the view model, which prepares data from the model to be displayed by the view.

Once the application starts, the main app coordinator is instantiated. The coordinator's responsibility is to manage navigation between screens. In the beginning, the coordinator instantiates the main menu screen with all its dependencies, such as the view model, and presents it to the user. Any input from the user that affects the navigation flow is directly forwarded to the coordinator, who decides how to handle it. It is important to note that the views are not coupled with each other and do not participate in the navigation flow directly; instead, they let the coordinator decide how to proceed. Each coordinator implementation must conform to the `Coordinator` protocol,

that is shown in Listig 10. The protocol prescribes a start method, a navigation controller, and an array of child coordinators that are commonly used in applications with a deeper navigation hierarchy.

Once a coordinator starts, it begins to manage the navigation flow. The Racing Dashboard application has only one coordinator, the `AppCoordinator`. In this case, one coordinator is sufficient, due to the trivial flow of the screens. Data from one screen to another are passed through the coordinator, and there is no backward data passing functionality; hence no closure callbacks or delegates are necessary.

```
1  protocol Coordinator {
2      // Array of child coordinators
3      var childCoordinators: [Coordinator] { get set }
4      // Navigation controller that is being used by the coordinator
5      var navigationController: UINavigationController? { get set }
6      // Starts the coordinator
7      func start(in window: UIWindow)
8  }
```

Listing 10: Coordinator protocol

5.3 User Interface

The application's user interface is created with a programmatic approach, using Apple's native UIKit framework with the addition of SnapKit for defining layout constraints. Contraints can also be created using the native AutoLayout; however, SnapKit offers a shorter and easier syntax. The GUI layout is designed to be responsive; therefore, it fits various screen sizes, such as iPhone or iPad devices. It was decided not to implement light variant of the GUI due to the tendency to disturb the user while using the application during a race.

In addition to basic GUI elements, custom elements have been designed solely for the purpose of the Racing Dashboard application, such as a vertical selector, gradient button, panel card, progress indicator, bar gauge, and more. The implementation of these specific elements is described in the following sections.

5.3.1 Vertical Selector

The main menu of the application features a vertical selector that is used to select a combination of a dashboard type and a racing game. The design of the element is shown in Chapter 4. The `VerticalSelector` is customizable and easily reusable. The selector accepts an array of items that are of type `SelectorItem`, where each item features a title, subtitle, background image,

and a gradient. The logic behind the selector listens to the user input, precisely to a pan gesture on the vertical axis. Once the selector item is dragged from a defined boundary, the next item is displayed using an animation. If the pan gesture ends before reaching the threshold, the currently displayed item returns to its original position. Transitions between elements are accompanied by a haptic response using the `UIFeedbackGenerator`.

5.3.2 Gradient Button

Throughout the application, customized buttons are used with an animated background. It is implemented as a subclass of UIKit's `UIButton`. In addition to the native button, it adds a custom animated gradient background. The animation is based on changing the location of the gradient for each color. Furthermore, the highlighted and enabled states are implemented using a simple opacity adjustment.

5.3.3 Panel Card

PanelCard is yet another custom element that is being used by the main menu and the graphs screen. Each panel displays a title, a subtitle, and an icon. The `PanelCard` is customizable, reusable, and automatically adapts to almost any size. Implementation is done via subclassing UIKit's `UIView`. There is an additional feature of icon rotation when the user taps the panel card. The purpose of the animation is only to entertain the user.

5.3.4 Progress Indicator

The progress indicator is a more advanced element in terms of implementation. It features two states with an animated transition between them. The first state reflects a work in progress. It is basically a fraction of the circle created using the UIKit's `UIBezierPath`, which rotates around the center of the arc to indicate a work-in-progress state. The `CABasicAnimation` takes care of the animation logic. Once the state changes to the other option that usually stands for success, the graphics change to a checkmark symbol. Both lines of the symbol are animated using `CABasicAnimation`.

5.3.5 Bar Gauge

Within the modern dashboard, a custom bar gauge is used. The bar gauge is one of the simpler elements; however, it is an integral part of the dashboard. It can be set to a vertical or horizontal layout. The gauge bar is essentially a custom loading bar that can display any value in the range between 0 and 1. This element is used mainly to display the position of the pedal or the level of gas. Implementation is done via subclassing `UIView` and changing size of a colored layer in relation to the current value to be displayed.

5.3.6 RPM Gauge

A significant part of the modern dashboard is occupied by a circular gauge, a custom progress indicator, which is similar to the already established bar gauge. There are two key differences between the circular gauge and the bar gauge. The circular gauge is created by a 270 degree fraction of a circle. In addition, it features a gradient that changes constantly in relation to the displayed value. Each layer in the gauge is created using a `CAShapeLayer` and `UIBezierPath`. The animation of progress is achieved using a mask layer rather than resizing the layer itself.

5.3.7 Other Elements

The application features many other custom elements, that will not be described in more detail. Custom elements are used so often due to more advanced GUI design, and the chosen programmatic approach of GUI creation supports custom elements even more. It is appropriate to amend that the application uses the theme pattern to provide consistent colors and font elements.

5.3.8 Documentation

Within the scope of this project, no extensive documentation was created. This decision is based on the fact that the result is not a library or application that will be utilized by other developers. Nevertheless, Xcode supports code documentation using specific markdown comments, which are cleverly displayed within the Quick Help and also as the description shown in the symbol completion. This feature was used to document the code. Furthermore, there is an option to generate the documentation based on the comments.

Chapter 6

Testing

Testing is an important part of the application development process. In this Chapter, testing is divided into four sections, namely, performance testing, unit testing, GUI testing, and practical testing. This multi-step testing process ensures the quality of the application. In addition, the practical testing section includes a video attachment that shows the in-game use of the application.

6.1 Performance Testing

An important aspect of the Racing Dashboard application is the networking functionality and, especially, the performance. Part of the networking design process aimed at a well-performing solution; therefore, the SwiftNIO library was chosen. Fast communication via the UDP protocol is crucial to ensure the highest possible data flow that provides a seamless rendering experience.

6.1.1 Testing Environment

The performance testing process took place on an Xbox One X, which came out as a more powerful Xbox One version in 2017 [45]. The racing game used during the testing was Forza Horizon 5. The Xbox device was connected to the router through cable and Wi-Fi. Both connection methods showed the same results. The device that ran the Racing Dashboard application was an Apple iPhone 12 Pro connected to a router via Wi-Fi. The connection between Xbox and iPhone was mediated by a Zyxel VMG3312-T20A router that offers a Wi-Fi standard 802.11 b/g/n with speed of up to 300 Mbps.

6.1.2 Testing Results

The UDP networking service receives data at a rate determined by the racing simulator. For example, Forza Horizon 5 can send up to 60 packets per second. The service receives all packets from the network and throttles the

data flow by a given value. The key parameter is defined in milliseconds and specifies how frequently the networking service data are passed further to the application. Testing was performed to determine the appropriate interval. The results are shown in Table 6.1. The table covers each tested interval, usability, the number of received packets per second, and a note. Usability was divided into *Good*, *Usable*, and *Unusable* segments, where *Good* represents the best usability, and conversely, *Unusable* represents the worst. This scale is subjective. An optional note captures additional observations.

Table 6.1: Networking performance testing

Interval	**Usability**	**Packets**	**Note**
500 ms	Unusable	1-2/s	Very unresponsive
200 ms	Unusable	3-5/s	Unresponsive
100 ms	Usable	5-7/s	-
50 ms	Good	10-13/s	-
No restriction	Unusable	18-22/s	Render issues

After reviewing the results of the tests, intervals greater than 100 ms were easily detected by the human eye in a negative manner. Without any flow restrictions, the GUI struggled with rendering glitches. The interval of 100 ms was usable, although the interval of 50 ms was noticeably smoother without any rendering glitches; as a result, the interval of 50 ms was chosen to be the most suitable.

6.2 Unit Testing

Unit testing is a testing approach in which individual units or modules are tested separately. Unit testing is the first level of the testing hierarchy, followed by integration testing, system testing, and acceptance testing. A single unit may be a method, function, procedure, or something else. Unit testing is carried out during the development phase and ensures that each unit of the software works as intended [46].

The Racing Dashboard application features unit tests for different modules, such as helpers, formatters, or decoders. XCTest framework was used for unit testing. Xcode reports test coverage to be 24,6% for the Racing Dashboard application. Examples of the unit tests implemented can be seen in Listing 11. The `testTimeFormatter()` test ensures a correct formatting from seconds in floating format to a time string represented in mm:ss:ms format. Furthermore, the `testHEXUIColor()` tests custom extension for `UIColor`, that are used for creating colors from hexadecimal and RGB formats. The default implementation of the `UIColor` does not support constructors for hexadecimal representation of color, and does not accept RGB in the common range of one byte per color. Therefore, custom extensions were introduced and unit tested.

Both tests shown in Listing 11 use a `XCTAssertEqual()` asserts, that tests if two expressions have the same value.

```
// Formatter tests
class FormatterTests: XCTestCase {
    // Test time formatter
    func testTimeFormatter() {
        let time = 128.285
        let formatted = Formatter.secondsToTimeString(time)
        XCTAssertEqual(formatted, "02:08:28")
    }
     // ...
}
// UIColor extension tests
class UIColorExtensionTests: XCTestCase {
    // Test conversion helper (hex, rgb)
    func testHEXUIColor() {
        let hexColor = UIColor(hex: 0xfcba03)
        let rgbColor = UIColor(red: 252, green: 186, blue: 3)
        XCTAssertEqual(hexColor, rgbColor)
    }
    // ...
}
```

Listing 11: Unit tests example

6.3 GUI Testing

The primary goal of GUI testing is to validate the features and performance of the application according to the requirements. GUI testing checks the functionality of the application by analyzing the interface that is visible to the user, including elements such as screens, menus, labels, buttons, and many more [47].

For the Racing Dashboard application, a manual testing approach was used. This testing method is based on manual use of the application. Each functional requirement was manually tested and modified according to the results.

6.4 Usability Testing

Usability testing is based on evaluating the application by testing it with the representative users. The Racing Dashboard application was tested with five

users covering all the use cases within the tested scenarios. Xbox One X with Forza Horizon 5 game installed was used in order to fully test the connection process. Table 6.2 shows a list of testers, consisting of a codename, age and a level of experience with racing simulators. The level is described as *Novice*, *Proficient*, and *Expert*, where *Novice* has the least experience and conversely *Expert* is experienced the most.

Table 6.2: Usability testing users

Tester codename	Age	Racing simulator experience
Tester A	27	Expert
Tester B	24	Novice
Tester C	24	Proficient
Tester D	21	Expert
Tester E	28	Novice

6.4.1 Testing Procedure

The testing procedure covered 3 different scenarios, where each scenario was designed to represent one or multiple use cases of the application. With this approach, each use case was tested by the users. Used testing scenarios are the following:

- **TS1: Selecting game and dashboard type:** The user is instructed to select Assetto Corsa as the game type and oldschool dashboard as the dashboard design. After selecting these parameters, the user proceeds to the connection screen. Once the user reached the connection screen, he is instructed to return back to he main menu.
- **TS2: Connecting to a racing simulator** The user is instructed to select Forza Horizon 5 as the game and modern dashboard as the dashboard design. Next, the user starts the connection process and establishes the connection between the application and Forza Horizon 5. The user is assumed to use the connection guide functionality. After setting up the connection, the user is instructed to launch the dashboard. Finally, the user returns to the main menu.
- **TS3: Reviewing race statistics:** The user is instructed to open the screen with race statistics. Next, the user is supposed to select the oldest record. Once selected, the user is instructed to display values for gas, brake a gear usage. After reviewing the values, the user returns to the main menu.

After finishing the testing procedure, every tester was asked the following questions:

- **Q1:** Did you encounter any problems during the execution of the scenario?
- **Q2:** Did you find the process to be clear and straightforward?
- **Q3:** Is there anything you would improve?

6.4.2 Results

A summary of the testing results is covered in this section. Most of the testers have completed the scenarios successfully with only some minor issues.

The first scenario (*TS1*) was successfully executed by each tester. Most testers did not encounter any problems and considered the process straightforward and clear. *Tester B* would improve the vertical selector by utilizing the chevrons to act as navigation buttons in addition to the gesture navigation. It was observed, that *Tester E* tried to tap the selector item, although the vertical selector does not respond to tap gestures.

The second tested scenario (*TS2*) was the most extensive. Each user successfully selected the instructed racing game and dashboard design. Once reaching the connection screen, the majority of the testers started interacting with the connection guide. *Tester C* did not take advantage of the guide. Each tester filled the given parameters into Forza Horizon 5 settings menu and successfully established a connection. After proceeding to the dashboard, 2 out of 5 testers were struggling with leaving the dashboard screen. It was unclear, that a double-tap gesture serves as a return action. All the testers were satisfied with the connection guide and considered it to be very clear. *Tester B* was observed to be struggling with navigation within the Forza Horizon 5 menu, although the tester declared, that the menu of Forza Horizon 5 seems confusing. All testers, except *Tester D*, would improve the return option from the dashboard, for example by introducing an overlay guide that would appear by the first launch of the application. On the other hand, *Tester D* claimed, that a double-tap gesture was the first thing that occurred to him and did not find it confusing.

The last tested scenario (*TS3*) was considered straightforward by all the testers. Each user found the graphs button very quickly due to the color contrast. Navigation between the records was executed by every tester with no issues. Each tester listed the displayed values as instructed with no hesitation. *Tester A* and *Tester C* did not interact with the graph, for example by swiping or zooming in. The remaining testers automatically started to interact with the graph without being instructed to do so.

After reviewing the testing results, a few changes are suggested. The issue addressed by every tester was the unclear return action from the dashboard itself. This could be solved by introducing an overlay guide that appears on the first launch of the dashboard. Furthermore, the chevrons of the vertical selector could be utilized to serve as a second selection method.

6.5 Practical Testing

The Racing Dashboard application has been tested in practice on a real racing simulator rig. The game used for the tests was Forza Horizon 5. The testing was based on the use of the application in a race. As a result, a video footage was recorded that contains a screen capture of the game and the Racing Dashboard application. The racing simulator rig features the following components:

- Racing seat CZC.Gaming Centaur
- Thrustmaster TMX PRO steering wheel
- Thrustmaster T3PA pedals
- Custom 3D printed handbrake
- Xbox One X
- Samsung TV UE55NU7093, 55”

From a subjective point of view, the application felt responsive and was able to completely substitute the in-game HUD. A screenshot of the footage can be seen in Figure 6.1. In addition, the complete footage can be found as an attachment to the enclosed CD.

Figure 6.1: Practical testing in Forza Horizon 5

Conclusion

In this work, various racing simulators were analyzed, including their respective communication methods for data output. Existing alternatives to the Racing Dashboard application were listed and thoroughly analyzed. Based on the communication methods of the chosen racing simulators, a networking layer was created to provide a well-performing communication process. On the basis of the analysis, an iOS application was designed, including self-designed GUI wireframes.

The result is a full-fledged mobile application that is used to display various data from racing simulators. It features an intuitive GUI and uses the UDP protocol as a foundation for communication. The application supports both iPhone and iPad devices running iOS 15. The process of establishing a connection is simple and intuitive. Race data recording is a vital feature that allows the user to review each race and use it as a basis for self-improvement.

This project aimed to create a fully functional mobile application that could be used in racing simulator rigs. When designing the architecture of the application, attention was paid to making the application easily extensible in the future.

Further development of the application could consist of extending the supported games and adding various predefined dashboards, as well as a fully customizable versions of dashboards, including features such as modifiable shift lights, configurable layouts, various themes, and many more. The application could be further extended to support the OBD2 adapters, which would extend the scope of the application to real cars. With some additional work, the application could be published on the App Store and used to improve the racing experience of racing simulators and even real vehicles.

Bibliography

1. *Apple Inc. Human Interface Guidelines* [online] [visited on 2022-01-10]. Available from: https://developer.apple.com/design/human-interface-guidelines/.
2. *Market research future: Racing Games Market Research Report* [online] [visited on 2022-01-15]. Available from: https://www.marketresearchfuture.com/reports/racing-games-market-9560.
3. *Statista: Racing Games* [online] [visited on 2022-01-15]. Available from: https://www.statista.com/outlook/dmo/app/games/racing-games/worldwide.
4. *RealDash: Supported ECUs and Games* [online] [visited on 2022-01-12]. Available from: http://realdash.net/support.php.
5. *RaceDash: Games* [online] [visited on 2022-01-12]. Available from: https://www.racedash.app/games/.
6. FALL, K.R.; STEVENS, W.R. *TCP/IP Illustrated*. Addison-Wesley, 2011. Addison-Wesley professional computing series, no. sv. 1. ISBN 9780321336316. Available also from: https://books.google.cz/books?id=X-l9NX3iemAC.
7. LAMMLE, T. *TCP / IP*. Wiley, 2017. ISBN 9781119472704. Available also from: https://books.google.cz/books?id=oDw7DwAAQBAJ.
8. *User Datagram Protocol* [RFC 768]. RFC Editor, 1980. Request for Comments, no. 768. Available from DOI: 10.17487/RFC0768.
9. *Apple inc. iOS and iPadOS usage* [online] [visited on 2022-03-20]. Available from: https://developer.apple.com/support/app-store/.
10. *Apple, inc. Apple Reinvents the Phone with iPhone* [online] [visited on 2022-04-02]. Available from: https://www.apple.com/newsroom/2007/01/09Apple-Reinvents-the-Phone-with-iPhone/.

11. *Apple inc. Identify your iPhone model* [online] [visited on 2022-03-19]. Available from: https://support.apple.com/en-us/HT201296.
12. *Apple inc. iPhone models compatible with iOS 15* [online] [visited on 2022-03-19]. Available from: https://support.apple.com/guide/iphone/supported-models-iphe3fa5df43/ios.
13. *Apple inc. iPhone SE - Technical Specifications* [online] [visited on 2022-04-01]. Available from: https://support.apple.com/kb/sp738?locale=en_US.
14. *Apple inc. iPhone 13 Pro Max* [online] [visited on 2022-03-18]. Available from: https://www.apple.com/iphone-13-pro/specs/.
15. *Appleinsider: A brief history of the iPad, Apple's once and future tablet* [online] [visited on 2022-03-03]. Available from: https://appleinsider.com/articles/18/04/03/a-brief-history-of-the-ipad-apples-once-and-future-tablet.
16. *Apple inc. iPad models compatible with iOS 15* [online] [visited on 2022-03-20]. Available from: https://support.apple.com/cs-cz/guide/ipad/ipad213a25b2/ipados.
17. *Apple inc. iPad mini 4 - Technical Specifications* [online] [visited on 2022-03-18]. Available from: https://support.apple.com/kb/SP725?viewlocale=en_EN&locale=zh_CN.
18. *Apple inc. iPad Pro, 12.9-inch (5th generation) - Technical Specifications* [online] [visited on 2022-03-18]. Available from: https://support.apple.com/kb/SP844?locale=en_EN.
19. *Forza motorsport forums: Forza Motorsport 7 Data Out feature* [online] [visited on 2022-02-07]. Available from: https://forums.forzamotorsport.net/.
20. *Toptal developers: iOS User Interfaces: Storyboards vs. NIBs vs. Custom Code* [online] [visited on 2022-02-16]. Available from: https://www.toptal.com/ios/ios-user-interfaces-storyboards-vs-nibs-vs-custom-code.
21. *Cocoacasts: SwiftUI Fundamentals* [online] [visited on 2022-02-17]. Available from: https://cocoacasts.com/swiftui-fundamentals-what-is-swiftui.
22. *Apple inc. Xcode* [online] [visited on 2022-02-09]. Available from: https://developer.apple.com/documentation/xcode.
23. *Cocoapods: What is CocoaPods* [online] [visited on 2021-12-05]. Available from: https://cocoapods.org/.
24. *Swift.org: Package Manager* [online] [visited on 2022-02-07]. Available from: https://www.swift.org/package-manager/.

25. *Apple: Swift Package Manager Project* [online] [visited on 2022-03-26]. Available from: `https://github.com/apple/swift-package-manager`.
26. *Apple Inc. UIKit* [online] [visited on 2022-01-15]. Available from: `https://developer.apple.com/documentation/uikit`.
27. *SnapKit: SnapKit* [online] [visited on 2022-03-20]. Available from: `https://github.com/SnapKit/SnapKit`.
28. *Swinject: Swinject* [online] [visited on 2022-04-05]. Available from: `https://github.com/Swinject/Swinject`.
29. *Apple Inc. Combine* [online] [visited on 2022-01-20]. Available from: `https://developer.apple.com/documentation/combine`.
30. *Apple Inc. Introducing Network.framework: A modern alternative to Sockets* [online] [visited on 2022-03-15]. Available from: `https://developer.apple.com/videos/play/wwdc2018/715/`.
31. *Apple: SwiftNIO* [online] [visited on 2022-02-01]. Available from: `https://github.com/apple/swift-nio`.
32. *Netty project: Documentation* [online] [visited on 2020-04-06]. Available from: `https://netty.io/wiki/index.html`.
33. DOUGLAS, Aaron; MORA, Saul; MOREY, Matthew; REA, Pietro; TEAM, raywenderlich.com. *Core Data by Tutorials Third Edition: IOS 10 and Swift 3 Edition*. 3rd. Razeware LLC, 2016. ISBN 1942878265.
34. *Cocoacasts: Core Data Fundamentals* [online] [visited on 2022-01-03]. Available from: `https://cocoacasts.com/what-is-core-data`.
35. *Apple Inc. Core Data* [online] [visited on 2022-02-01]. Available from: `https://developer.apple.com/documentation/coredata`.
36. *PhilJay: MPAndroidChart* [online] [visited on 2022-01-14]. Available from: `https://github.com/PhilJay/MPAndroidChart`.
37. *Danielgindi: Charts* [online] [visited on 2022-07-03]. Available from: `https://github.com/danielgindi/Charts`.
38. *Apple inc. XCTest* [online] [visited on 2022-02-04]. Available from: `https://developer.apple.com/documentation/xctest`.
39. *Realm: SwiftLint* [online] [visited on 2022-04-05]. Available from: `https://github.com/realm/SwiftLint`.
40. GELATKA, Adam. *Mobilní scanner dokumentů* [online]. Praha, 2020 [visited on 2020-02-13]. Available from: `https://dspace.cvut.cz/bitstream/handle/10467/88172/F8-BP-2020-Gelatka-Adam-thesis.pdf`. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií. Supervised by Ing. Dominik VESELÝ.
41. *Khanlou: The Coordinator* [online] [visited on 2022-02-05]. Available from: `https://khanlou.com/2015/01/the-coordinator/`.

42. *Adobe XD Ideas: Mobile First Design Strategy: The When, Why and How* [online] [visited on 2022-01-21]. Available from: https://xd.adobe.com/ideas/process/ui-design/what-is-mobile-first-design/.
43. *James Shore: Dependency Injection Demystified* [online] [visited on 2022-03-16]. Available from: http://www.jamesshore.com/v2/blog/2006/dependency-injection-demystified.
44. *Apple Inc. Localization* [online] [visited on 2022-02-01]. Available from: https://developer.apple.com/documentation/xcode/localization.
45. *Windows central: Xbox One X tech specs* [online] [visited on 2020-04-06]. Available from: https://www.windowscentral.com/xbox-one-x-specs.
46. *Guru99. Unit Testing Tutorial: What is, Types, Tools and Test EXAMPLE* [online] [visited on 2022-04-07]. Available from: https://www.guru99.com/unit-testing-guide.html.
47. *Guru99: GUI Testing Tutorial: User Interface (UI) TestCases with Examples* [online] [visited on 2022-04-04]. Available from: https://www.guru99.com/gui-testing.html.

Appendix **A**

Acronyms

DI	Dependency injection
ECU	Engine control unit
FH5	Forza Horizon 5
GUI	Graphical user interface
HUD	Head-up display
IDE	Integrated development environment
RPM	Revolutions per minute
SoC	Separation of concerns
UDP	User datagram protocol
UI	User interface
UX	User experience
MVVM	Model-view-viewmodel

Appendix B

Contents of Enclosed CD

```
readme.txt ........................ the file with CD contents description
├── diag .......................................... the directory of diagrams
├── src ....................................... the directory of source codes
│   ├── impl ........................................ implementation sources
│   └── thesis .............. the directory of LaTeX source codes of the thesis
├── text .......................................... the thesis text directory
│   └── thesis.pdf ........................... the thesis text in PDF format
└── footage ................................... the video footage directory
    └── fh5Race.mp4 .................. Forza Horizon 5 testing video footage
```

Appendix C

Racing Dashboard Screenshots

A set of screenshots from the Racing Dashboard application taken on iPhone 12 Pro running iOS 15.4.1.

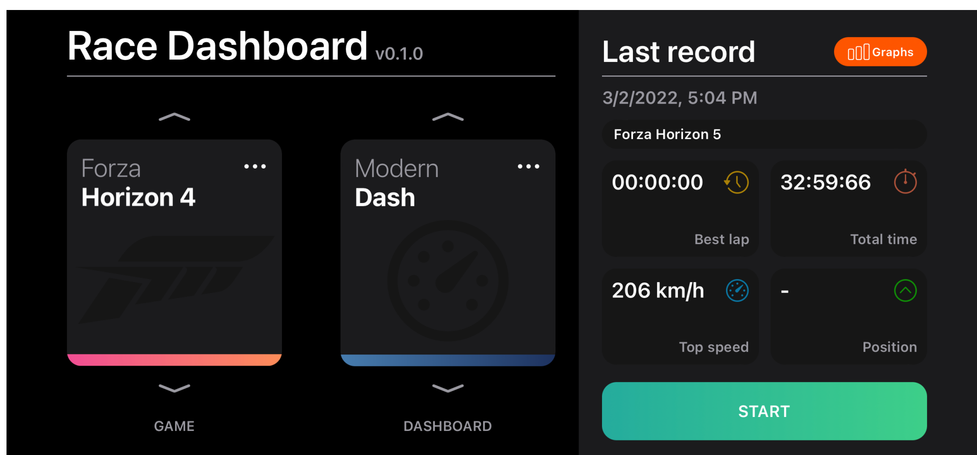

Figure C.1: Main menu screen screenshot

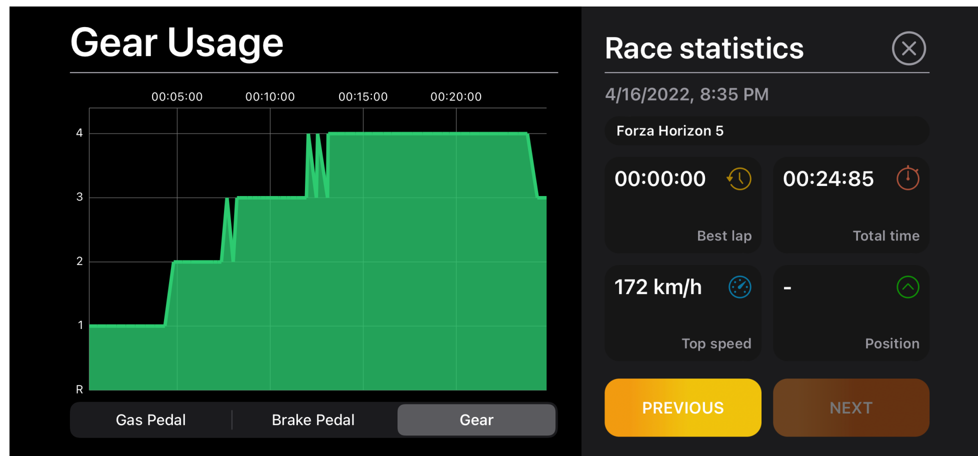

Figure C.2: Graph screen screenshot

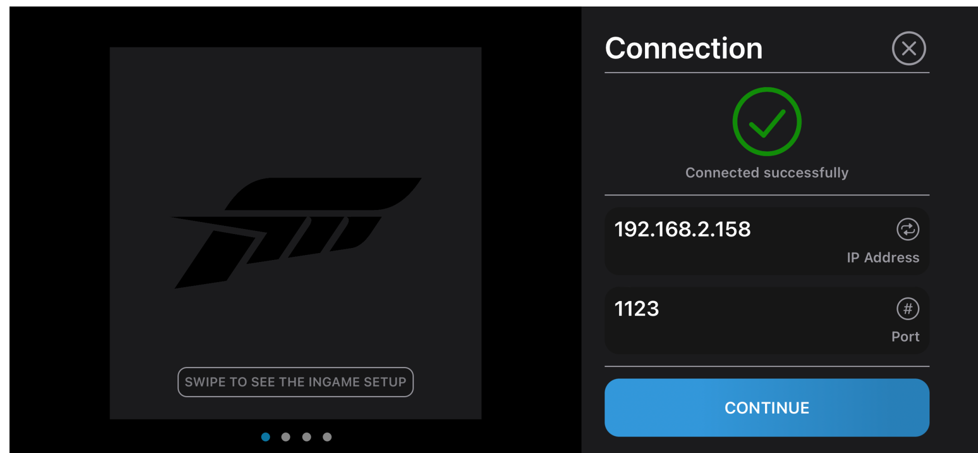

Figure C.3: Connection screen screenshot

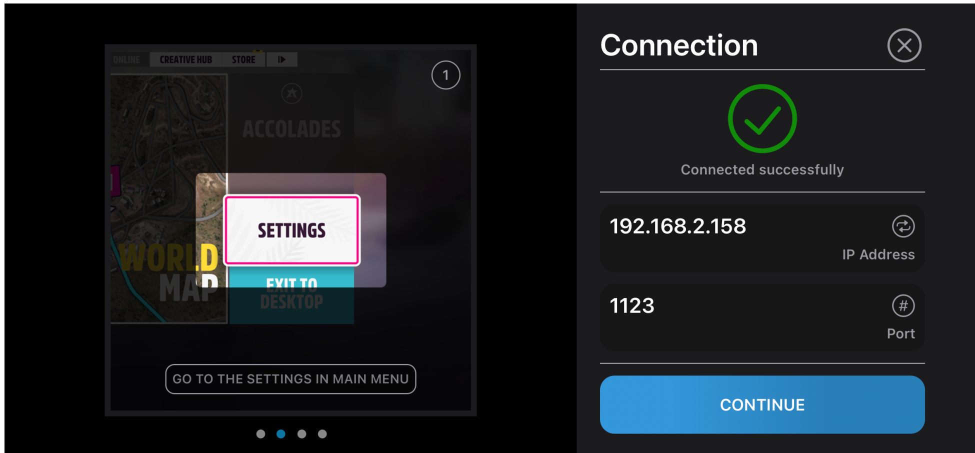

Figure C.4: Connection screen guided screenshot

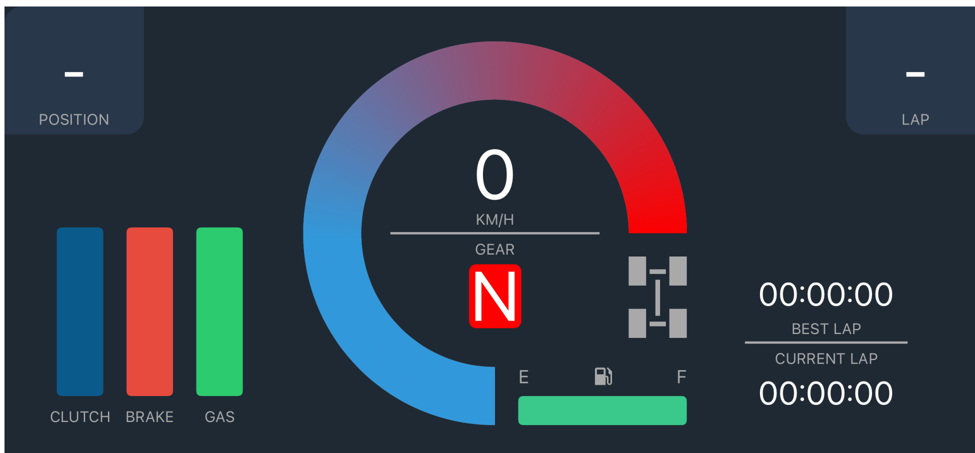

Figure C.5: Dashboard screen screenshot