Implementation

This chapter focuses on the development of the Virtual Rival Framework. At first the general architecture and the main components are outlined. After that, we discuss the structure of the developed modules and their functionality in detail. The modules have to fulfil the requirements discussed in chapter \ref.

# General Architecture

This section gives an overview over the main components of this project. Figure illustrates the components and how they work together.

**Game Server**

We use itch.io to {<https://itch.io/>} host our project. The website allows hosting games for free. Players can connect to the server and access the game client in their web browsers. The server manages the game data and synchronizes the actions of the players. We created a customised landing page for the players. It is a simple way to distribute the project.

**Amazon Mechanical Turk**

Amazon Mechanical Turk is a Amazon Web Service enabling in individuals to coordinate human tasks. Crowdsourcing has a dramatic impact on the speed and scale at which scientific research can be conducted \citep{Chandler}. \textcite{Lenz} showed that respondents recruited with Amazon Mechanical Turk are often more representative of the U.S. population than in-person convenience samples. The integration of Amazon Mechanical Turk allows us to perform easy and a low-cost field experiments.

**Database**

During the experiments the data from the questioners and the driver statistics is collected. We use two different data storage services to store user data.

FTP Server: The FTP Server is used to store binary data e.g. trajectories, time-steps. The data is enclosed in the message body and stored on the server.

Drive Platform (Google Drive): The Drive platform lets developers to open, import, and export native Google Docs types such as Google Spreadsheets, Presentations, Documents, and Drawings. We use the Drive API to access the data. Questionnaire data is directly transferred to Google Drive Forms, which allows storing data from online surveys. The data is requested in the Data Evaluation component.

**Player Client**

The client is accessed from the Game Server. It is available as Web Application and Desktop Application. The client was released with simultaneous support for Windows, macOS, and Linux. The Web Application supports all major browsers e.g. Firefox, Chrome and Internet Explorer. Other systems are not tested. The player client includes the race simulation and the questionnaires.

**Data Evaluation**

The Data Evaluation Tool is decoupled from the Player Client and Game Server. The Data Evaluation Tool is a standalone Python application. The player statistics are accessed from the database. The Data Evaluation Tool consists of a visualisation module and a statistic module. The visualisation module plots trajectories, lap-times-charts and personality graphs. The statistic module identifies trends using statistical data analytics methods e.g. mean values, standard deviations and hypothesis testing.

# D:\Lukas\Documents\Masterarbeit\Master Thesis\Visio\Design and Requiremnet\General Architecture.jpg

Conducting Clinical Research Using Crowdsourced Convenience Samples – Chandler

Evaluating Online Labor Markets for Experimental Research: Amazon.com's Mechanical Turk – Lenz

<https://developers.google.com/drive/api/v3/about-sdk> - Google Drive

# Race Simulation Design

Part of the project is to create an interactive 3D racing game. The game is rendered in the browser. Different browsers demand different standards. In order to have a stress-free transition between different platforms, we focused on a simple, plain design and the most trivial functions. This also helps to enhance performance. In order to make the entry point to the game as smooth and easy as possible we employed the conventional control system and traditional graphical visualization known from other racing games. The tutorial level is utilised the basic controls but also estimate the initial skill level. The difficulty is adjusted automatically when progressing across the laps. This allows for a flexible learning process, adjusted on the individual skill of the player.

# Race Simulation Design

Unity empowers game designers to make games. This section will explain the core concepts we used to create the Virtual Rival World. The environment is designed using obstacles and decorations. The game is rendered in the browser. Different browsers demand different standards. In order to have a stress-free transition between different platforms, we focused on a simple, plain design and the most trivial functions. This also helps to enhance the performance. Unity is structured in Scenes. A Scene is a unique level containing the environments or menus of a game. In general, we designed three different scenes types for our project:

**Instruction Scene**

The Instruction Scenes are the main tool to inform and advice players. The scenes are constructed out of four main unity components:

Image Component: The Image Component presents UI elements on the screen. The Image Components make out the background, sliders and panels in a scene. As source image, we use two-dimensional bitmaps which are often referred to as sprites in computer graphics. Image Components are integrated into a larger scene with other elements.

Button Element: The Button Element detects user input and triggers an event. The user input is trigger when hovering over or clicking on the Button Element. The Button Element integrates an Image Component to specify the background of the Button Element.

Text Field: The Text Field is a rectangle element displaying text on the screen. The font style can be changed as needed. We use *Arial* a classic san serif, which is characterized by simple formed and excellent graphic representability on screens (Arial).

Input Field: The Input Field is a specialisation of the Text Field, which makes the text editable. An event is trigger when the text content of the Input Field changes.

<https://docs.microsoft.com/de-de/typography/font-list/arial> (Arial)

Instructions can come in various shapes, depending on the task. We implemented three types of Instruction Scenes, with different functionalities: Basic Instruction Scene, Loading Information Scene and Individualised Information Scene. The scenes are constructed out of the four main unity elements from above. The Instruction Scene types are described in more detail below:

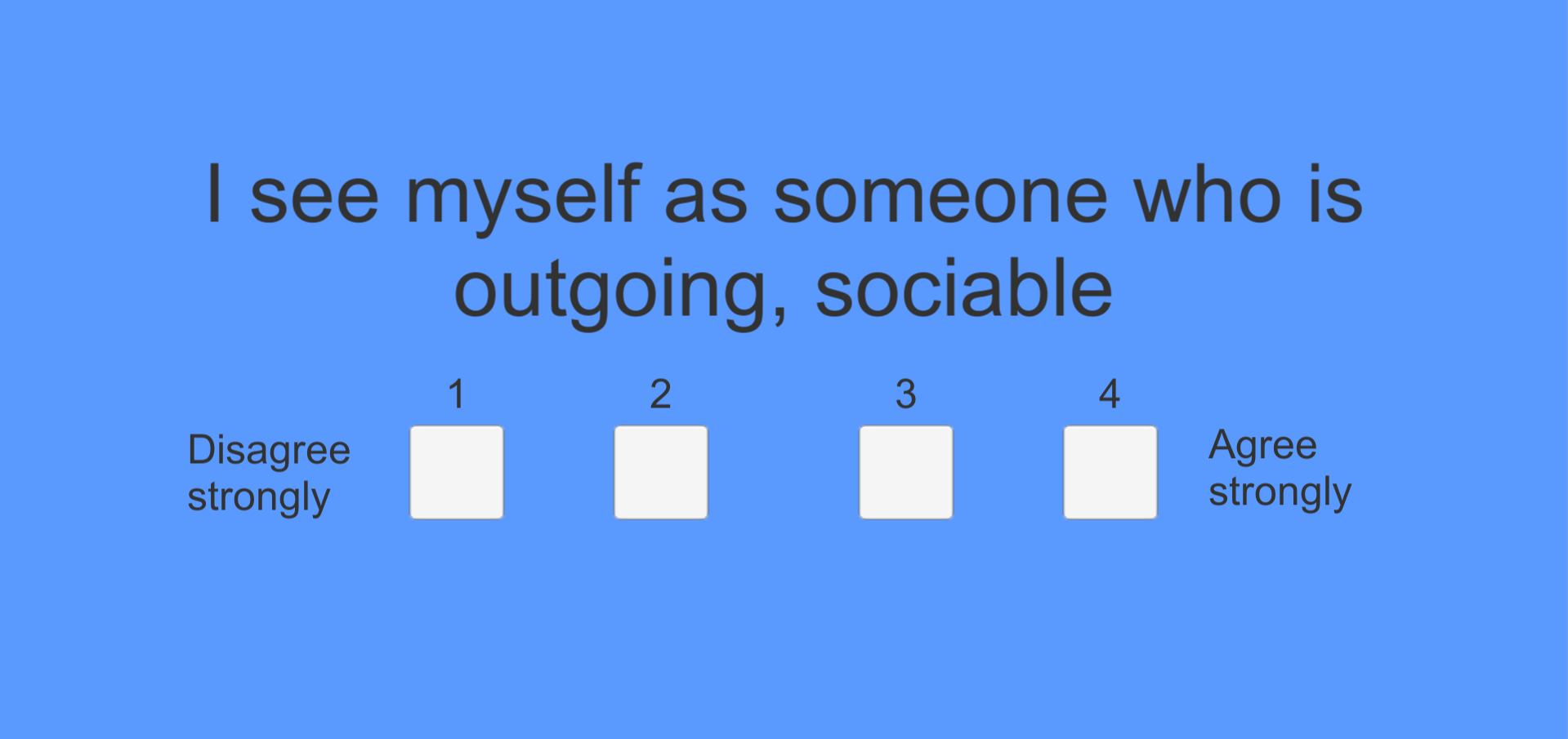
Basic Instruction Scene: The Basic Instruction Scene is constructed out of an Image Component for the background and one or multiple Text Fields. This scene type is used at the beginning to give the user instructions how to do the questionnaires and the controls of the car on the race track. Figure \ref exemplifies the Basic Instruction Scene used at the beginning of the racing segment.

Loading Information Scene: The Loading Information Scene is visually similar to the Basic Instruction Scene. In the background data is transmitted asynchrony to and from the server. The data is player information, questionnaire records and player trajectories. Figure \ref illustrates the Loading Information Scene between questionnaires.

Individualised Information Scene: The Individualised Information Scene retrieves individual player information from the server and displays it. Figure \ref illustrates the Individualised Information Scene to retrieve the unique player id.

**Questionnaire Scene**

The developed questionnaire unit is a useful instrument to run questioners. One can think of many different questionnaires to integrate in this project. The *Questionnaire Scene* fits every type of self-reporting psychological questionnaires. In our case, the questionnaires are *Sensation Seeking* (see section) and *Big Five (see section)*. Figure shows the Big Five Questionnaire Scene.



The Questionnaire Scene is constructed out of three major components. The Question Module and the Range Module are view components that show the question and answer possibilities. The Choice Module and the Confirm Module are interactive components that let the user choose an answer. The main components of the scene are illustrated in Figure. The modules are specified in detail below:



Logic Module: The Logic Module includes the logic which selects the questions. The questions are randomized. The module tracks answered questions and uploads them to the server. The logic of the scene is specified in more detail in Section \ref.

Choice Module: The Choice Module consists of multiple radio buttons. Only one answer can be selected. The Choice Module triggers the Confirm Module.

Range Module: The Range Module specifies the answer possibilities of the question. The basic question answers range from 1 - 5 or “Disagree Strongly” to “Agree Strongly”.

Question Module: The Question Module shows the current question. The current question is retrieved from the logic module.

Confirm Module: The Confirm Button is only visible when an answer is selected. Pushing the button triggers the Logic Module to select the next question.

**Race Scene**

The Race Scene is the central element of the project. We want to create a physical realistically driving simulation. To ensure convincing physical behaviour, the car must accelerate correctly and be affected by collisions and gravity. Unity’s built-in physics engine provides components that handle the physical calculations. Using the build-in Unity components we create objects that behave in a realistic way. The concrete movement is controlled by scripts. A typical object in unity holds both build-in physics components and scripts. Section \ref discusses the scripts controlling the objects. The main graphics primitives in Unity are 3D Meshes. Unity offers various components to import and render meshes, trails or 3D lines. Meshes make up the largest part of our 3D world. The main Unity components we use for every object (<https://docs.unity3d.com/Manual>):

* Texture: Wrap around the object to decorate the surface.
* Material: Defines how the object is displayed. The properties if a Material are determined by the Shader in use. A Shader is a special program that combines texture and lightning information to generate pixels.
* Transform: Defines position, rotation and size of the object.
* Colliders: Are used to detect environment collisions. Can be generated out of the mesh data.

Every object in the Race Scene integrates the main Unity components. All race tracks are building out of similar building blocks. They only differentiate in the race track layout. The main building blocks of the race scene are:

* Race Car: Figure illustrates the model used for the race car. To optimise performance we reduced the number of vertices. The main components that specify the physical behaviour of the car are the Rigidbody and WheelCollider component. The Rigidbody handle’s motion using the Unity physics engine (<https://docs.unity3d.com/ScriptReference/Rigidbody.html>). The body will regulate forces e.g. gravity, acceleration and react to collisions. The WheelCollider is a special build-in collider for grounded vehicles(<https://docs.unity3d.com/Manual/class-WheelCollider.html>). It integrates wheel physics and friction. By adding spring and damping forces we created a realistic suspension model for the car. Section~\ref introduces the WheelDrive script which controls the driving physic properties: motor torque, brake torque and steer angle.
* Roadway: The roadway and the Race Car are the main gameplay elements. The race track is assembled out of basic road elements e.g. straights, curves and s-curves in different sizes. Guide rails are placed on the edges to prevent players to leave the track. Crashing into the rails triggers a collision in the WheelDrive script (see \ref).
* Environment: The Environment presents visually appealing surroundings. Unity’s Terrain system allows us to create vast landscapes(<https://docs.unity3d.com/Manual/script-Terrain.html>). We modified the height map and applied rocky surface textures to create a valley. The scene is completed by adding rich vegetation: grass, trees, brushes and flowers. To create a wind effect we added a Unity wind zone. Trees within a wind zone bend in a realistic fashion and create a natural movement pattern among the trees. Figure~\ref illustrates the final scene design in Unity. To enhance the rural feeling we added nature noises in the background.
* Cameras: Unity uses cameras to render the scene. It is one of the most essential components in Unity. We use three cameras for our Race Scene. The main camera is attached to the vehicle and gives us a third person view over the scenery. The rear camera is similar to the main camera but points backwords. Figure shows how the rear camera is rendered on top of the main camera to create a rear-view mirror effect. The map camera is an orthographic camera with a top-down view of the scene. Normally, things far away are rendered smaller. An orthographic camera has no diminishing perspective. The frustum is straight and front and back have the same size. We use the orthographic camera to generate an isometric projection of the scene which visualises the tree-dimensional world in two dimensions as a map.
* Collider, Wheel collider
* Physic
* Rigidbody
* 3D Models – vertices, Finish Line, Car, Checkpoints
* Wind
* Checkpoint
* Environment Grass Tree Water Bush Terrain
* Street / Rails
* Light
* Cameras – Map, Rear-view mirror
* Sound



# Race Simulation Architecture

This section will explain the core concepts we used to create the gameplay mechanics. The previous section discussed Unity’s built-in Components. We implemented scripts to create our own gameplay features. Unity scripts are small programs which trigger events, modify component properties and handle user input. We use C# to write scripts. The script is connected to the Unity engine by deriving form the build-in unity class MonoBehaviour. Listing shows a blueprint of a unity script. There are two main functions defined inside the class. The code inside the update function is called each frame. We use update to create movement, trigger actions and handle user input. The start function is called when the scene is instantiated. We use start to initialise components and variables.

using UnityEngine;

using System.Collections;

public class CustomScript : MonoBehaviour {

void Start () {

// Used for initialization

}

void Update () {

// Called once per frame

// Custom behavior

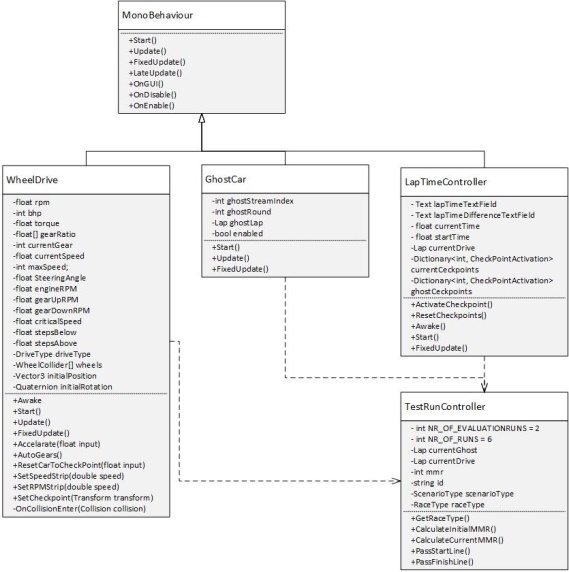
}

}

Anatomy of a Unity script file

https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html

In the following section, class diagrams and descriptions of the most important classes can be found. For the sake of readability, some classes have been omitted from the class diagrams. The class diagram in Figure shows the most important classes. The main classes are:

**

**WheelDrive**

The WheelDrive script defines the driving behaviour of the car. It is directly attached to the race car as well as a WheelCollider and Rigidbody component. The Unity WheelCollider component implements basic graphical wheel representations and roll mechanics. The Rigidbody connects the vehicle to the physics engine (see section). Listing shows the essential parts of the script. The script defines the car handling characteristics. In the initialisation phase we setup the wheels. During the continuous updates handle the user input. Pressing the acceleration button applies a positive force in forward direction. Depending on the DriveType we apply the force on two or all of its wheels simultaneously. The force is calculated with Formula. Changes in gearing are important when looking at torque, because the gears act as torque multipliers. The engine rpm is calculated to evaluate the engine sound. Engine rpm is correlated to the pitch.

Torque M(Nm)

Power P (bhp)

Gear ratio alpha ()

M\_vec = P \* alpha.

User steering modifies the steering angle of the two forward wheels. When pressing the reset button the car will be placed on the last checkpoint. The *WheelDrive* script also tracks the collisions while driving.

**GhostCar**

The *GhostCar* script uses a data from previous runs to create an opponent. The *TestRunController* selects suitable opponents. It is attached to the ghost car model. Algorithm~\ref{} illustrates the process. At the start we place the ghost car at the start line. In every frame we update the position by interpolating to the next location of the recorded ghost opponent.

Get Rotation & Position Data From Server

Initialise Ghost Car position and rotation

For every timestep

1. getPositionForTimestep(timestep)
2. interpolate rotation & position

**LapTimeController**

The *LapTimeController* traces the lap time and the activation of checkpoints. The timer is stopped at the finish line. During the run the timer is visualised on the upper edge of the screen. At the checkpoint we compare the timer with the opponent’s timer. The difference in time is displayed to inform the player and give feedback.

**TestRunController**

The TestRunController tracks the internal state during the run. The class tracks the players’ progress and skill level. The current skill level is updated after every lap. Since the real strength of a player is unknown to us, we have to estimate it by a rating. The Elo system is a rating system of competitive games. We modified the original system to fit the race game genre. Section~\ref describes the Elo algorithm in detail. The match between the player and the ghost can be approximated with the formula:

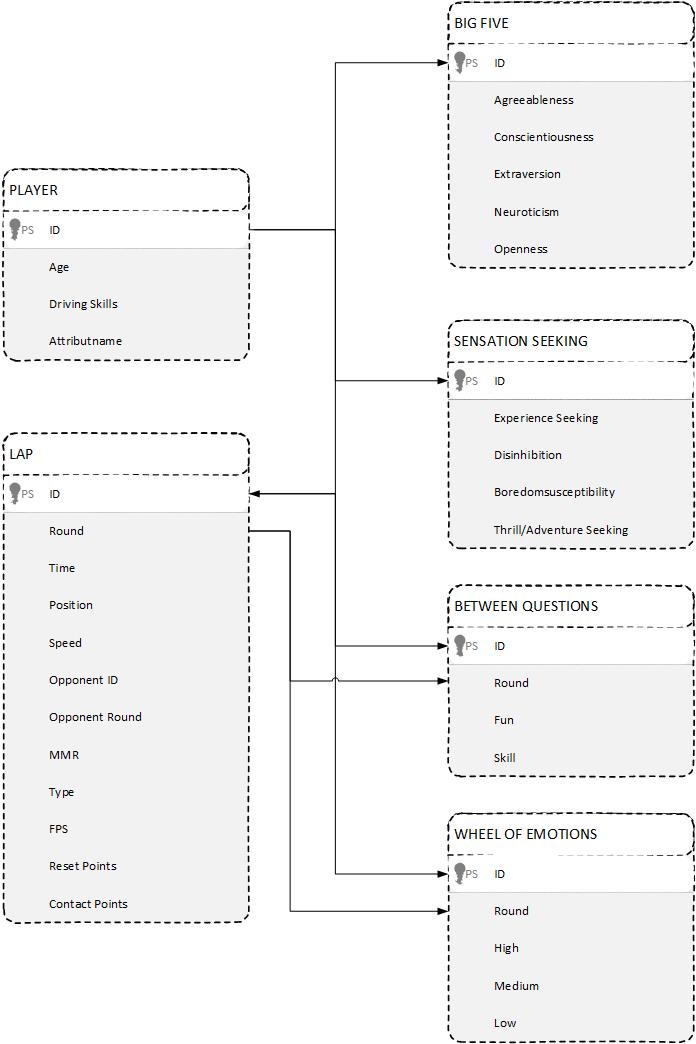
The expected score for the player to win the race based on the unknown strengths for player (RP) and ghost (RG). An expected score of 1 predicts a win and a loss by 0. Lambda describes the spread of the ratings. We chose Lambda with 400 which results in the win probability distribution in table \ref.

|  |  |
| --- | --- |
| Rating difference (RP-RG) | Estimated win probability for player |
| 0 | 0.5 |
| 50 | 0.42 |
| 100 | 0.35 |
| 200 | 0.24 |
| 400 | 0.09 |
| 800 | 0.01 |

After a race we have to update the strength. To update the player strength RP we use formula \ref. The race result is modelled in SP (WIN = 1, LOSS = 0). The expected result and the estimated strength are known quantities. The factor k bounds how fast algorithm involves. We set k to 128. A large k allows us to quickly find the correct skill niveau, but we lose precision.

# Data model

The run data is persistently stored in a cloud database. We use google forms to save the questionnaire data. The data is directly transferred from Unity to the server using the UnityWebRequest. The data can be visualised in the web. For the analytics tool in Section \ref, we use the Anaconda oauth2client library to access the data. The database model is shown in Figure \ref. During the driving course we save position and rotation information every 25ms. Additionally we save driving faults like accidents and reset points. The questionnaire data is saved in separate tables. The ID uniquely identifies a player. ID and Round form the foreign key for the questionnaires after each round.



# Analysis tool

The analysis tool visualises and interprets the stored data (see section). It’s a standalone Python application, independent from the other components. We use the data science platform *Anaconda* and the open source package management system *Conda* to manage libraries and dependencies.The main libraries are:

* NumPy: NumPy is one of the most fundamental Python libraries. It provides data structures for big, more dimensional matrices and efficiently implements numerical calculations.
* Pandas: Pandas provides high-performance data structures for data analysis.
* Scipy: The scipy.stats module contains a large number of probability distributions and a wide range of statistical functions.
* Matlibplot: Matplotlib is a basic Python 2D plotting library. We generate plots, histograms, errorcharts, and scatterplots using Matplotlib.
* Seaborn: Seaborn is a data visualization library for making statistical graphics. It provides convenient views for complex datasets.

*Anaconda* allows us to analyse data with scalability and performance with *NumPy* and *Pandas*. The results are visualised using *Matlibplot* and *Seaborn*. Visualisation is a central part of understanding data. We use Seaborn onto data snippets to produce informative plots. Figure \ref shows the composition of the analysis tool. The tool is constructed out of three components:

Reader Module: The Reader Module accesses the data from the server and parses them into suitable data container. Every data type has a special reader implementation e.g LapReader, QuestionsReader. The Graph Module and the Statistic Module manipulate the data.

Graph Module: The *Graph Module* visualises the data using *Matplotlib* and *Seaborn*. We plot histograms and scatterplots to analyse driving data.

Statistic Module: To find trends inside the data we use the statistics module of SciPy. The Statistic Module measures the linear relationship between two datasets. We calculate correlation coefficients and use p-value testing for non-correlation.

Library

<https://docs.scipy.org/doc/scipy/reference/stats.html>

<https://matplotlib.org/>

<https://seaborn.pydata.org/>

# Summary

In this chapter a brief overview of the structure was given and the modules that were developed for this work were presented. In addition, we introduce the used libraries and tools. We modelled a 3D race environment with integrated questionnaire using Unity. The implemented Analysis Tool allows us to find trends and correlations inside the data. This section focused on the implementation details.

First, we discussed the graphical design of the Virtual Rival Framework in Unity. The functionality is split across multiple Unity scenes. A scene represents an independent level in Unity. We differentiate three types:

* Instruction Scene: Show information to players e.g. controls, instructions, loading progress.
* Questionnaire Scene: Integrates different questionnaires to measure Enjoyment, Motivation and Education.
* Race Scene: Each scene includes different race tracks, race cars and terrains.

Secondly, we go into the implementation of gameplay for each scene. Gameplay in Unity is mainly constructed with predefined components and individual scripts. The race behaviour and physics calculation is specified to fit web application with restricted resources. To simulate a realistic race we implemented a ghost car using the Elo algorithm to match the players’ skill.

Third, we introduce the data management part of the Virtual Rival Framework. Data collected during the race and from questionnaires is directly uploaded to the cloud. Google services are used to store and access the data. The focus is on privacy, consistency and reliability.

The created 3D environment should give an example of what types of race scenarios are possible with the implemented tools. It is a first prototype and many more applications are thinkable. In the next step we have evaluated the Virtual Rival scenario with a user study in which we have collected data on three research focuses with the help of a set of standardized questionnaires. The next chapter will discuss this evaluation and its results.