UNIVERSIDADE D
COIMBRA

Alexandra Eugénia Ferreira Garcia Monteiro

SERIOUS GAME FOR STATIC BICYCLE

Dissertation in the context of the Master of Science in Electrical and Computer Engineering, Specialization in Computers, Subspecialization in Computational Learning, supervised by Prof. Dr. António Paulo Mendes Breda Dias Coimbra and Prof. Dr. João Paulo Morais Ferreira and presented to the Faculty of Sciences and Technology, Department of Electrical and Computer Engineering.

July 2023

Serious Game for Static Bicycle

Supervisors:

Professor Doutor António Paulo Mendes Breda Dias Coimbra (ISR, DEEC)

Professor Doutor João Paulo Morais Ferreira (ISR, ISEC)

Jury:

Prof. Dr. Mahmoud Tavakoli

Prof. Dr. Mário João Simões Ferreira dos Santos

Prof. Dr. António Paulo Mendes Breda Dias Coimbra

Dissertation submitted in partial fulfilment for the degree of Master of Science in Electrical and Computer Engineering.

Coimbra, July 2023

This project was developed with the cooperation of:

Universtity of Coimbra

1 2 9 0

UNIVERSIDADE D
COIMBRA

Electrical and Computing Engineering Department

deer.uc

DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA
E DE COMPUTADORES

Instituto de Sistemas e Robótica

INSTITUTE OF SYSTEMS AND ROBOTICS
UNIVERSITY OF COIMBRA

Acknowledgements

This dissertation was conducted under the guidance of Prof. Dr. António Paulo Coimbra, Prof. Dr. João Paulo Ferreira and Prof. Dr. Manuel Crisóstomo, to whom I express my sincere gratitude for support and orientation throughout this project. To my friends, a big thank you for sparing time to help me with my tests and for the company when I needed them. But most of all, to my family for their love and understanding throughout all the ups and downs. The authors acknowledge Fundação para a Ciência e a Tecnologia (FCT) for the financial support to the project UIDB/00048/2020.

Resumo

A capacidade motora dos membros inferiores é crucial para a realização de tarefas diárias. Com o crescente número de população envelhecida, obesidade e doenças relacionadas, existe cada vez uma maior necessidade de desenvolvimento de novos métodos para melhorar a mobilidade da população. A reabilitação física desempenha um papel crucial na recuperação destas doenças, embora a falta de consistência possa prejudicar o processo de recuperação. A criação de ambientes cativantes tem-se revelado uma ferramenta importante na consistência ao longo do percurso de reabilitação. Considerando que a Plataforma de Desenvolvimento em Tempo Real, Unity, permite o desenvolvimento de projetos 3D em tempo real, esta tese apresenta um método inovador para o desenvolvimento destes ambientes. O objetivo principal deste trabalho é desenvolver um jogo sério para uma bicicleta estática motorizada, que possa ser utilizado durante o processo de reabilitação, permitindo ao utilizador uma experiência imersiva e a possibilidade de escolher entre diferentes modos de exercício adequados às suas necessidades individuais. Inicialmente, foi elaborado um documento de requisitos técnicos, onde foram apresentadas as principais diretrizes para o jogo sério. Posteriormente, foi criado um modelo da bicicleta em Blender, para atingir uma interação mais realista. Adicionalmente, foi também desenvolvida a possibilidade de alterar o campo de visão, bem como três modos de jogo diferentes. Por fim, procedeu-se à criação de menus bem como, a possibilidade de regular as configurações durante o decorrer do exercício. Devido a circunstâncias externas, os testes com a bicicleta estática real não puderam ser realizados. No entanto, um comando para video jogos foi utilizado para simular as várias entradas da bicicleta estática com o objectivo de garantir que o jogo pudesse ser testado e avaliado para atender aos requisitos desejados.

Abstract

The motor capacity of the lower limbs is essential to perform daily tasks. With the rising numbers of demographic ageing, obesity and related diseases, there is an increasing need for developing new methods to improve the mobility of the population. Physical rehabilitation plays a crucial role in the recovery from these illnesses, although a lack of consistency can be detrimental to the recovery process. The creation of captivating environments has been shown to be an important tool in maintaining consistency throughout rehabilitation. Considering that Unity Real-Time Development Platform allows its users to build real-time 3D projects, this thesis presents an innovative method for the development of these environments. The main objective of this work is to develop a serious game for a motor-assisted static bicycle, that can be used during the rehabilitation process, allowing the user to have an immersive experience and the possibility of choosing between different modes of exercise adequately to their individual needs. Initially, a technical requirement document was elaborated, where the main guidelines for the game were presented, afterwards, a model of the bicycle was created in Blender, to achieve a more realistic interaction. Additionally, the possibility to alter the field of view was also developed, as well as three different game modes. Finally, the game menus were created as well as the possibility of adjusting the settings during the course of the exercise. Due to external circumstances, tests with the real static bicycle could not be performed. However, a game controller was used to simulate the various inputs of the static bicycle to ensure that the game could be tested and evaluated to meet the desired requirements.

Keywords

Serious game; Rehabilitation; Static bicycle; Unity; Physiotherapy; Mobility.

Contents

List of Acronyms

2D	Two-Dimensional
3D	Three-Dimensional
CVA	Cerebrovascular Accidents
FND	Motor Functional Neurological Disorder
CF	Cystic Fibrosis
PEP	Positive Expiratory Pressure
PC	Personal Computer
UCD	User-Centered Design
LWJGL	Lightweight Java Game Library
VR	Virtual Reality
USB	Universal Serial Bus
HID	Human Interface Device
SDK	Software Development Kit
SUS	System Usability Scale
UI	User interface
PS3	PlayStation 3

List of Figures

List of Tables

1 Introduction

1.1 Context and Motivation

The increasing life expectancy and sedentary lifestyle have caused an impact on the overall health of the population [1], [2].

As a result, motor impairments are becoming common, many of which, diminish an individual's quality of life, causing an inability to carry out various tasks essential to maintaining an independent lifestyle[3].

Some of these pathologies cause a discrepancy in the mobility of each limb, making it necessary to train each member individually, to achieve an effective outcome.

One of the most effective treatments is physiotherapy, this form of treatment has been shown to be a great tool for gaining back mobility and overall improvement of the life quality of these patients. [4], [5].

Despite the many positive aspects of physiotherapy, there are still some setbacks in this form of medical rehabilitation. Taking into account that for physical rehabilitation to be effective patients need various sessions during a long period of time, this type of commitment can be hard to achieve for many individuals.

A critical factor in improving exercise adherence is through an interactive environment, for example, a video game. The development of an interactive 3D game environment, using Unity's software, that can address patients' needs is not only beneficial to keep patients motivated but has also the advantage of offering different modes of exercise tailored to each patient's unique condition.

1.2 Main Goals of this Dissertation

The main goal of this work is the development of a 3D serious game for a motor-assisted static bicycle, that allows the user to choose between three different modes and adjust values

such as resistance and weight according to their characteristics. More specifically, the steps to achieve this goal were:

- Conduct research on the current state of the art, in the fields of physical rehabilitation and serious games,
- Incorporate into Unity a playStation 3 (PS3) control as an input to better simulate the static bicycle responses,
- Develop a virtual representation of a bicycle that behaves similarly to a real-world bicycle, excluding the aspect of falling to the sides when standing still,
- Develop game menus that allow the patient to save his data and adapt the game variables to their specific case, for example, weight, height and level of resistance,
- Demonstrate that the mechanics of the serious game function correctly, and that the interface is user-friendly.

1.3 Document Overview

This document is structured as follows. A literature review of mobility, physiotherapy and previous projects on this theme is performed in Chapter 2. Afterwards, in Chapter 3, the development of this work is explained in detail, the choice of software, along with different steps to reach the final outcome, that take into account the different modes of the motor-assisted static bicycle. The development is tested in Chapter 4, which presents the experimental setup and results. Finally, Chapter 5 presents a reflection upon the overall success of this work, proposing a number of potential lines of future work in order to improve the current prototype.

2 Background

2.1 Main causes of mobility issues and rehabilitation

It is important to understand what are the main causes of mobility issues in the population, contributing factors, and how these problems can be overcome. Mobility issues have a major impact on a person's quality of life [3], [6]. People with these disabilities are more likely to be unemployed, have higher healthcare-related costs, and present a greater risk of hospitalizations [7].

The ageing of the population, obesity and sedentarism are possible factors contributing to mobility disability[8], [9]. These factors do not cause mobility problems directly, however, they place a person at a higher risk for accidents and developing diseases, compromising his ability to move.

Some of the diseases that can affect mobility are:

- Neurological disorders that leave patients with motor sequels such as cerebrovascular accidents (CVA) [4], Parkinson's disease [10], multiple sclerosis, motor functional neurological disorder (FND) a common cause of persistent and disabling neurological symptoms[11],
- Musculoskeletal health conditions, for example, Rheumatoid arthritis, Osteoarthritis, Carpal tunnel syndrome, Fibromyalgia and many others [12],
- Conditions that can lead to amputations, type 2 diabetes, type 1 diabetes [13], peripheral arterial disease, malignant tumours, trauma [14],
- Long periods of hospitalization and bedrest orders, especially in older generations [15].

The main form of treatment for these conditions is physiotherapy. This type of therapy has shown promising results in improving mobility, for example in the case of a CVA, even if rehabilitation is started late after a stroke, mobility improvements are achieved [16]. Patients that suffered from trauma also showed an improvement in mobility after rehabilitation [5].

Other conditions such as Ankylosing Spondylitis [17], trauma [5], chronic multiple sclerosis [18] also have shown improvements from physiotherapy.

This work can be beneficial in prolonging physiotherapy's benefits, one of the biggest downsides of this type of rehabilitation is the amount of time required to achieve results, and the consistency needed to maintain said results[19]. Video games promote treatment adherence and motivate the patients to continue the treatment in the medium or long term.[20].

2.2 Related work

To improve the development of this work, research was conducted on serious games.

According to Tarja Susi et al. [21] the definition of a serious game is "a digital game that is used for purposes other than mere entertainment", this concept is applied to a broad spectrum of application areas, e.g. healthcare, military, educational and industry. One of the goals of this type of game is to motivate the patients during the physical therapy process, as the treatment can be repetitive and tedious.

For example "A Serious Game for Rehabilitation of Neurological Disabilities" [22], indicates that, typically, the exercises recommended by physiotherapists for patients that suffer from motor disabilities caused by neurological issues, are "repetitive and boring", due to the long duration of the and the repetitive nature of these physical therapy treatments, which leads to a lack of motivation and abandonment of the treatment. Consequently, pleasant environments are designed to help motivate patients to persevere until the end of their therapy, which raises their odds of success in recovery.

Figure 2.1: Scenes from the serious game [22].

In this particular case, research was done on the most common and transversal exercise, for various neurological diseases to then chose the ones that were able to be monitored by image processing techniques, the game was then designed to be adapted to these types of exercises and made simple enough so that the players do not lose interest, due to difficulty or lack of understanding.

The game is built in Unity 3D which allows integration with the Microsoft Kinect software development kit, permitting the control of the Microsoft Kinect sensor used to monitor and detect physical therapy movements. This project was developed in Unity 3D, and the programming languages used were Javascript and C#. The graphic modelling scenarios, characters and objects were developed in Unity 3D, Blender and iClone.

This game was also developed with physiotherapists in mind, therefore a three-dimensional structure was created for health professionals and researchers (see Fig. 2.2). All information shared between the three-dimensional structures is stored in a remote server and is accessed through the Internet. The first area consists of the game itself, from the selection of players to the levels and rankings of players. In the back office, oriented to health professionals, the physiotherapist creates the profile of patients and defines rehabilitation for each one of them. The health workers can see the evolution of their patients, and record notes, among other functionalities. Furthermore, the back office for researchers is objectively prepared to make a statistical analysis of the evolution of each patient.

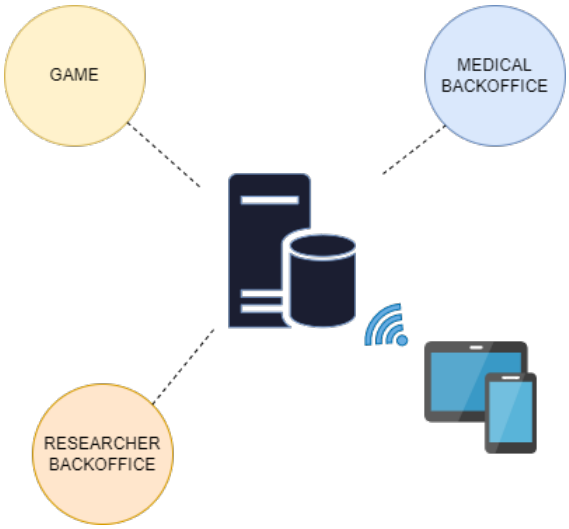


Figure 2.2: The three-dimensional scheme of the application *Adapted from [22].*

A similar concept was implemented in [23], an effective method of removing mucus build-up in the lung of sufferers of chronic lung diseases such as cystic fibrosis (CF) is positive expiratory pressure therapy (PEP). Despite this, this form of treatment can be difficult to

use on children and teenagers, causing stressful situations and conflict. The hardware used is the PEP device, widely available. The innovation created was the connector to the PC and the games that can be controlled through air pressure (see Fig.2.3). Furthermore, the platform created can track engagement and patient scores, providing additional motivation for players to keep participating in each game through the competitive ranking of the team's performance. In addition, the games are able to collect important information such as the duration of gameplay and litres of air exhaled by each user. This type of data can be useful for:

- Monitoring patient's evolution throughout therapy, and consequently optimising their treatment,
- Academic research,
- User insight to enhance the platform.

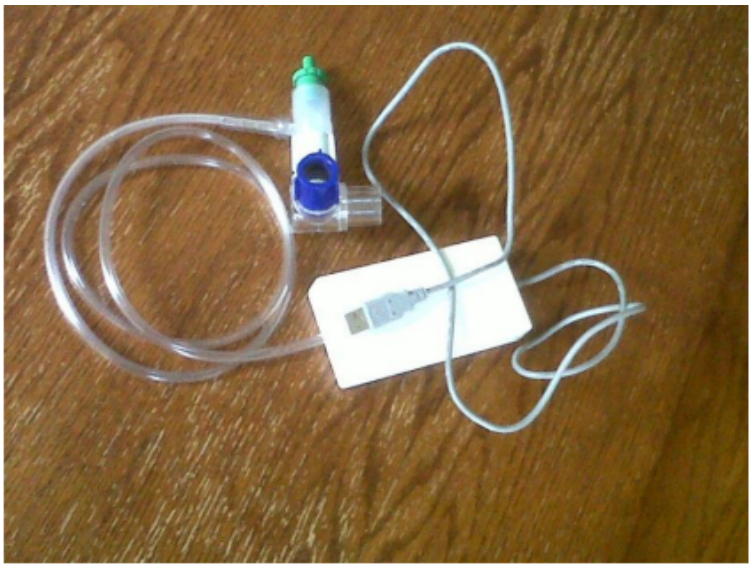


Figure 2.3: PEP device with connector for the PC [23].

Similar projects were created to aid in treating patients that suffered from a stroke. One such example is [24]. Following a CVA patients frequently relate partial paralysis or balance-related problems. Usually, rehabilitation training relies on strength and balance exercises. As a result, balance plates were chosen as training devices. However, "patients often report lack of motivation, especially when training at home". The design of this game was based on a user-centred design (UCD) approach, which takes into account the development process around the end users, in this case, the patients. In addition, the physiotherapist's input was

added to better understand the needs and problems of the patients, and improve the overall outcome.

The software used was Java Server Faces, JavaScript and Slick2D, a 2D game library based lightweight java game library (LWJGL), developed under the BSD-License. This framework offers interfaces for OpenGL, OpenAL and includes libraries for game controllers. The hardware used to complement the game was the Wii Fit Plus package including the Wii Fit Balance Board. Because it is based on a relatively stable and fixed low platform, and it is easy to set up.

This application can help patients throughout the rehabilitation process, because its development is centred around the patient's individual needs and his relationship with the physiotherapist. Furthermore, because of computer-supported monitoring during physiotherapy, this application is also a helpful tool for physiotherapists alike, by delivering objective feedback on a patient's progress throughout the rehabilitation process. The "RehaLabyrinth-Patient" application is running locally on a computer. After the rehabilitation session, the collected data is sent to the web service and stored in the database. Moreover, the web service has an interface for various operations such as: "creating a report on the patient's rehabilitation progress", "Highscore list" and others. For the physiotherapist, there's a web interface named "RehaLabyrinthTherapist" that offers full access to all the database patient's stored data (see Fig. 2.4).

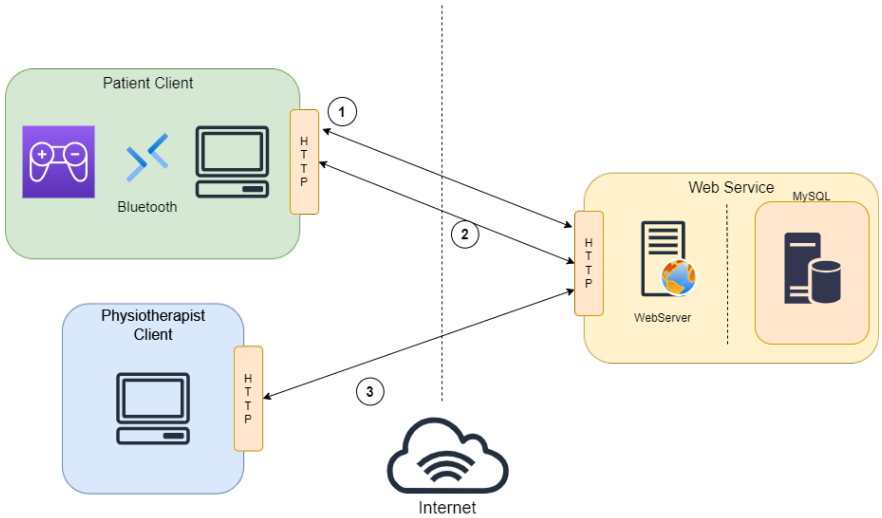


Figure 2.4: RehaLabyrinth Architecture, *Adapted from [24].*

The main goal of this application, "RehaLabyrinth" (see Fig. 2.5), is to overcome challenges caused by a lack of motivation during the exercises, by providing modifiable and individual setups of balance training, whilst offering a competitive environment by creating

the possibility of achieving bonus points upon mastering a variety of mazes. In doing so therapeutic tasks become more exciting for the patient, helping the patient achieve better results.

Figure 2.5: RehaLabyrinth Level Overview [24].

Finally, [25] focuses mainly on obtaining patient data such as biometric data, for example: Angle velocities, the distance between feet, left right feet usage frequency.

As well as medical records and scores during the serious game-based training, usage frequency and execution time for exercises.

One of the main benefits of this type of serious game is in the hardware used, Kinect v2 motion sensing sensor. This technology has a "higher accuracy in human body detection" making it a perfect candidate for physiotherapy scenarios. It can provide data associated with the patient joint position and training location. This type of information can be then analyzed in correspondence with the selected rehabilitation game. Furthermore, with the mobile application, connected to Kinect frameworks, the physiotherapist can follow their patients from anywhere, thus reducing the therapist's work and allowing the patients to train at home.

This serious game was developed using the Unity 3D game engine, to ensure that it was possible to build different virtual reality (VR) scenarios (see Fig. 2.6). For each type of patient, this gives the patient the possibility of following the movement of the action performed by an avatar. Or, depending on the type of rehabilitation, play the game that focuses on training individual body parts.

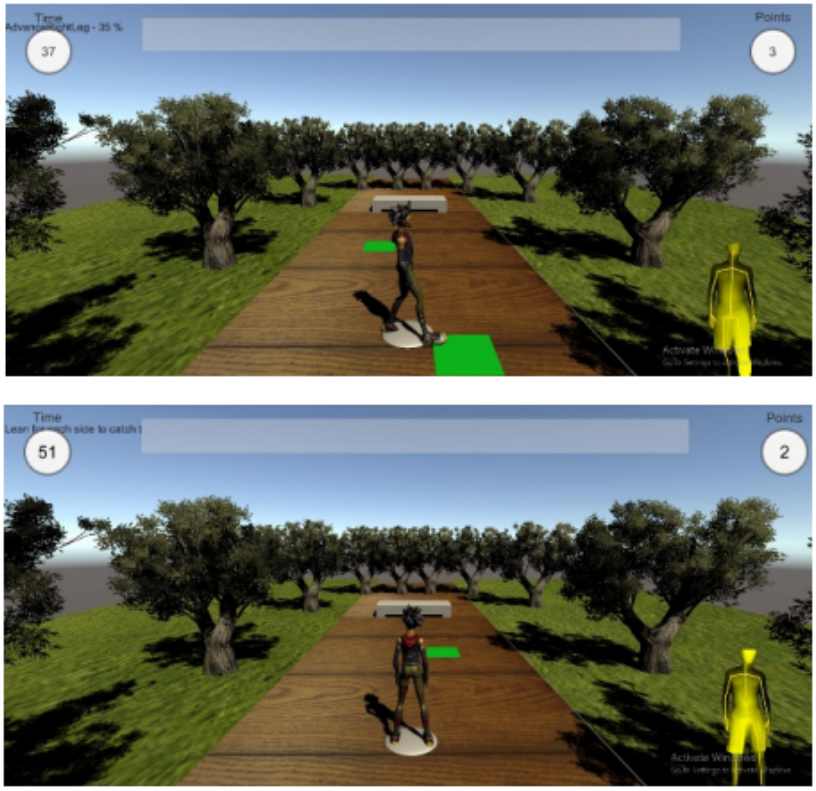


Figure 2.6: Serious game example for lower limb rehabilitation [25].

2.3 Game engines

In this section of the work, there will be a discussion about game engines, firstly, it is important to understand why these engines were created and how they work, before debating the advantages and disadvantages of each one.

In the early stages of the video game industry, each game had to be programmed from scratch due to hardware limitations. Later on, with the advancement of hardware and the need for faster game development cycles, the concept of game engines emerged. A game engine is a reusable software layer that allows the separation of general game concepts from game components (levels, graphics, etc.).

The first video games created were coded from scratch as a "singular entity, with no relation to previous games", for example game platforms, such as the Atari or the Commodore at the beginning of the 80's, created their games this way. It wasn't until the mid-90's that the first video game engines started to appear, due to the popularity of 3D first-person shooter games like "Doom" and "Quake". By 1998, releases such as "Quake III" and Unreal were already being designed, completely separating the engine from the content, as well as, separating game-specific rules and data from their basic and abstract concepts.

The definition of a game engine is a software framework primarily designed for the development of video games. Its main goal is to "abstract common video game features allowing for code and game asset reuse in different games". As a result, these are some of the typical functionalities found in a game engine:

- Input handling,
- Scene graph,
- Rendering for 2D and 3D graphics,
- Game loops,
- Sounds,
- Scripting,
- Memory management,
- Process threading,
- Animation for models,
- Physics engine, with collision detection.

Game engines have been keeping a broad definition, in the sense that their features and workflow vary from one another. Most industry-level engines are greatly flexible and powerful, making it possible to create any type of game across a number of different platforms. Besides making games available across other platforms, another major goal is optimising developers' productivity by using high-end programming languages such as C#, Java and Python (see Table 2.1).

Table 2.1: Summary of game engine survey[26].

Game Engine	Game gender & Renderer	Publishing Target	Starting Price for Commercial Usage	Scripting Language	Supported Platforms
Unity	Generic 2D / 3D	Windows; OSX; Linux; PlayStation; Xbox; Wii; iOS; Android; BlackBerry; Windows; Phone; Browsers	Free	C#	Windows; OSX; Linux
Unreal Engine	Generic 2D / 3D	Windows; OSX; PlayStation; Xbox; iOS; Android; Browsers	Free	C++	Windows; OSX; Linux
CryEngine	Generic 3D	Windows; OSX; Linux; PlayStation; Xbox; Wii;iOS;Android	€9.90/month	C++ or Lua	Windows; OSX; Linux
Torque	Generic 2D / 3D	Windows; OSX; iOS	Free	TorqueScript, Similar to C++	Windows; OSX; Linux
Flixel	Generic 2D	Flash; iOS; Android	Free	ActionScript 3	Windows; OSX; Linux
GameMaker: Studio	Generic 2D	Windows; OSX; Linux; iOS;Android;Windows	Free or $99.99	Visual Programming Language or Game Maker Language	Windows; OSX; Linux
Construct 2	Generic 2D	HTML5	€99.99	Visual Programming Language and JavaScript	Windows; OSX; Linux
Stencyl	Generic 2D	Windows; OSX; Linux; iOS;Android;Flash	Free, 99/*year*, or 199/*yeat*	Visual Programming Language	Windows; OSX; Linux
Visionaire Studio	2D Adventure	Windows; OSX; Linux; iOS; Android	€49	Not required	Windows; OSX; Linux
eAdventure	2D Adventure	Windows; OSX; Linux; Browsers	Free	Not required	Windows; OSX; Linux
RenPy	2D Visual Novel	Windows; OSX; Linux	Free	Python	Windows; OSX; Linux
RPG Maker	2D RPG	Windows	$69.99	Not required	Windows; OSX; Linux
Phaser	Generic 2D	Browsers	Free	JavaScript	Windows; OSX; Linux
Turbulenz	Generic 2D / 3D	Browsers	Free	JavaScript	Windows; OSX; Linux

In conclusion, after reviewing the literature on diseases that compromise mobility, other serious games and available game engines, the Unity game engine was chosen to develop this work, mainly because of its ability to create 2D and 3D games alike, the wide range of platforms it can develop games for and the ability to export for fifteen different platforms and all major browsers. In addition, Unity also has a notably good price point, being free to use for indie developers. The ability to create simple scenes via its visual editor and continuously integrating new features made this engine a great tool.

3 Game Development

3.1 Unity's Input System

The motor-assisted static bicycle is being improved by another student, because of this, it is not available for testing. Therefore, a PS3 game controller was used to simulate the various inputs of the system. Unity supports two separate input systems, an older version, which is built-in to the editor and is called the "Input Manager", and a newer one called "Input System Package".

This later version allows the usage of any kind of input device to control Unity content, and it was created with the intent to replace "Unity's classic Input Manager". As a result, the installation of the newer input system was crucial for this project since it allows the integration of the PS3 controller [27], as well as the possibility of supporting any device which implements the universal serial bus (USB) human interface device (HID) specification. Even if devices don't have specific layouts implemented in the input system it is possible to manually remap the controls [28].

The input system offers four different ways of workflow, and the developer can choose the one that better suits the type of project being developed. To better understand how the different workflows operate, firstly an overview of the concepts and terms used to describe them is required [29]. These basic concepts and terms refer to the steps in the sequence of events that occur when a user sends input to a game or app. The Input System offers features which implement these steps, or they can be altered by the developer to suit their project better (see Fig. 3.1).

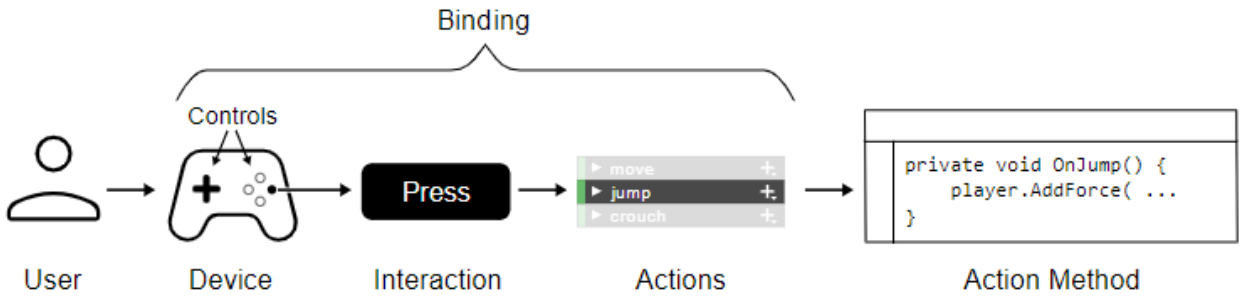


Figure 3.1: Sequence of events that occur when a user sends input [27].

The four types of workflows are "Direct", "Embedded Actions", "Action Asset" and "PlayerInput Component". The "Direct workflow" is the easiest and fastest way to set up the input system, although it is not recommended for projects with multiple types of input or platforms. Because "Direct workflow" reads values directly from the devices in the C# script (see Fig. 3.2), unlike the other options available, it is harder to keep code organised since "there is no abstraction between the code and the values generated by a specific device".

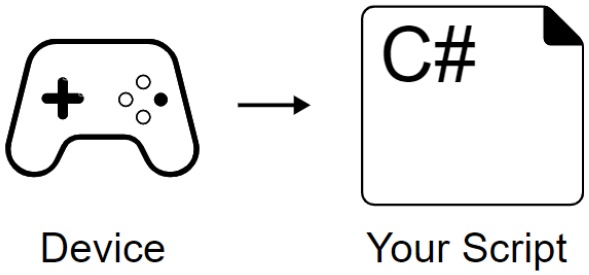


Figure 3.2: Workflow - Direct [27].

"Embedded Actions" create an additional layer of abstraction between the methods and the device bindings compared to Direct workflow, as it allows the programmer to create an "InputAction class". In this way there's no need to specify what each control should do in the code explicitly, instead, this is defined within the "InputAction class", where the actions are bound to the controls and respond to the values or states within the code. The actions display in the script's inspector, and allow their configuration in the editor (see Fig. 3.3).

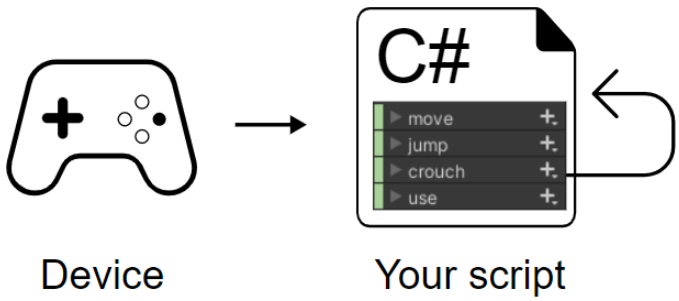


Figure 3.3: Workflow - Embedded Actions [27].

The third way of using the input system is "Actions Asset" [30]. This was the workflow chosen for this project. This way there's no need to define actions directly in the script. Alternatively, the script references an "Input Actions Asset" which defines each action. Consequently, the data that defines the actions are kept separate from the "GameObjects" which should respond to the actions. All action definitions are stored as a single asset file, separate from project scripts and "prefabs" [31] thus making it simpler to manage conceptually. This can be useful for bigger projects where multiple people need to work on different parts of the project. Compared to "Embedded Actions", "Actions Asset" allows the developer to group actions into Action Maps and Control Schemes.

Action Maps are a way to group related actions together, where each map relates to a different situation. For example, there can be an action map for character movement, with actions called "walk", "run", "jump", and "crouch". The Control Schemes, also defined in an Action Asset, allow the developer to specify which bindings belong to the control schemes that are defined. This is useful when a game has multiple input devices.

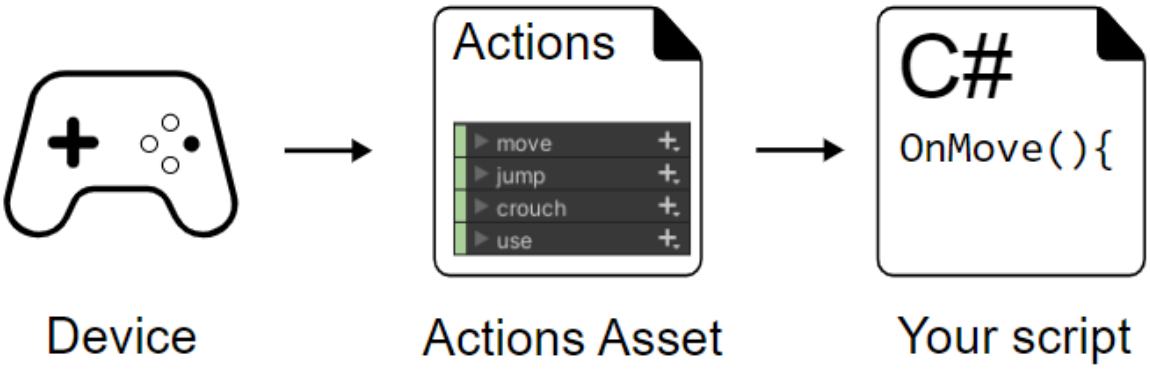


Figure 3.4: Workflow - Actions Asset [27].

Finally, the developer can use "Actions Asset and PlayerInput Component". This work-

flow provides the highest level of abstraction. The Player Input component allows the developer to make connections between the actions defined in an "Actions Asset" and C# methods in the "MonoBehaviour scripts", using the user interface (UI) [32] in the inspector. There's no need to write additional code to create these connections. This way the methods are called when the user performs an input action. Compared with the previous workflow code example, even though this method requires less code, it does require more setup in the Editor and could make debugging more difficult because the connections between the code and the action are not hard-coded.

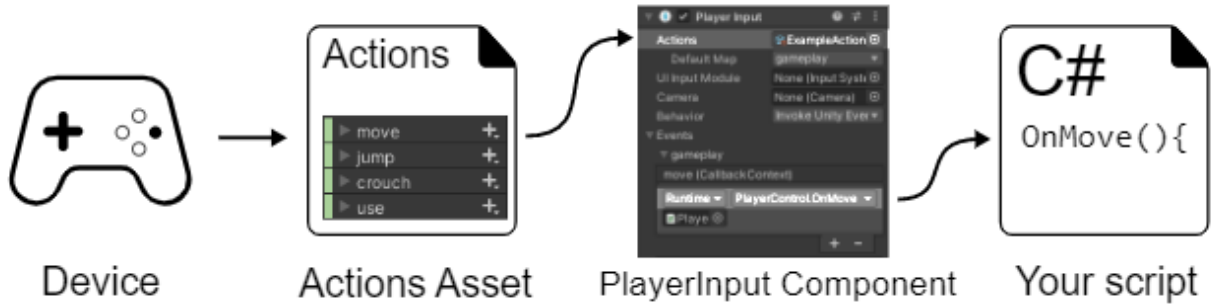


Figure 3.5: Workflow - PlayerInput Component [27].

In this project, an action map called "Gameplay" was created, (see Fig. 3.6). The actions "MoveBikeL" and "MoveBikeR", are bonded to the left and right thumbsticks, the y component of both the left and right thumbsticks represents the input force of the left, and right leg respectively and the x component of the left thumbsticks represents the direction in which the bicycle steering is headed (see Fig. 3.7). In addition a way for the user to view the scene around them was implemented, the actions intended for this purpose are "Xleft", "Xright", "Yup" and "Ydown" these are bound to the buttons on the right side of the controller, "Xleft" corresponds to Player can look to the left, "Xright" corresponds to Player can look to the right, "Yup" corresponds to Player can look to the up, "Ydown" corresponds to Player can look to the down (see Fig.3.7). Furthermore, a way for the player to stop the movement was added with the action called "Brake" and this is bound to the L1 button of the PS3 controller (see Fig. 3.7). All these actions are of the type "Value", this means that all the controls bound to an action are continuously monitored. Afterwards, the values from the control actuated to be the control driving the action, are reported in callbacks.

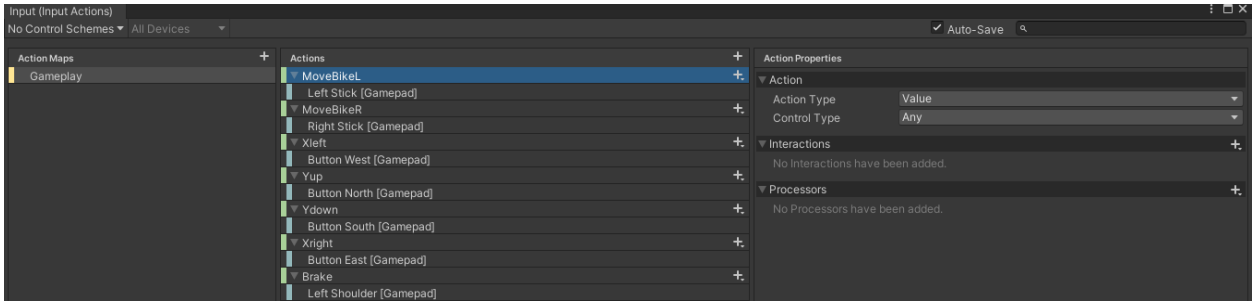


Figure 3.6: Action Map and respective actions.

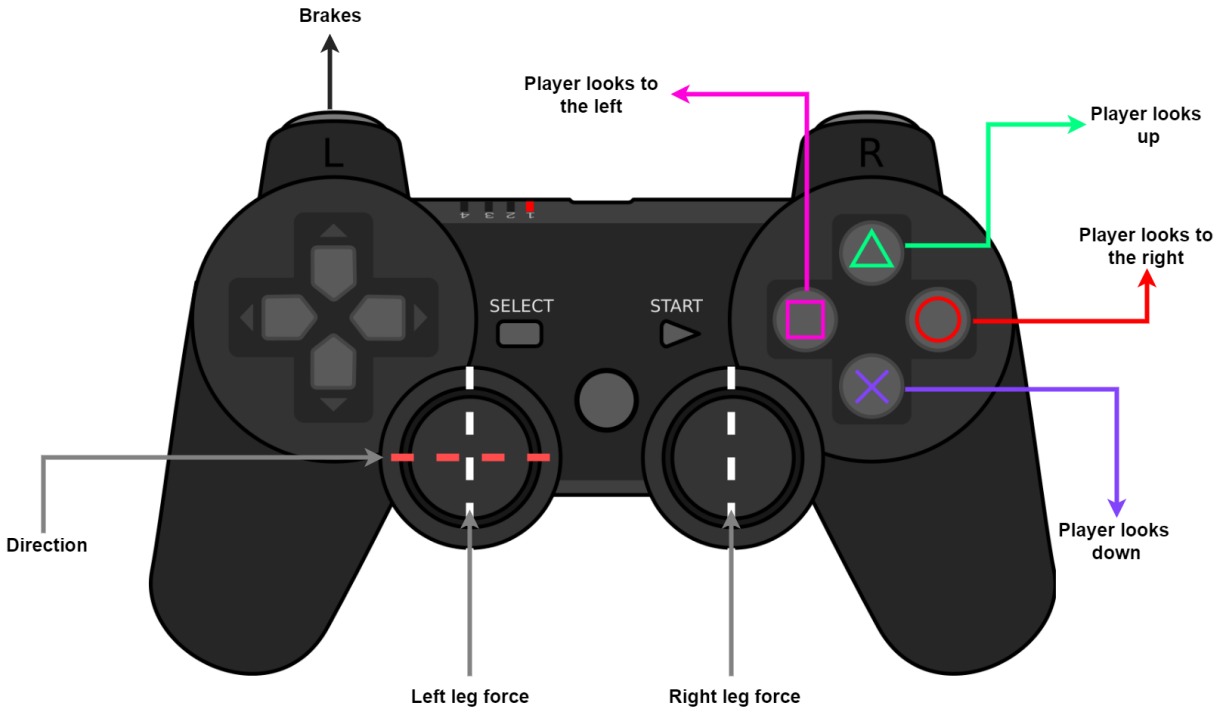


Figure 3.7: PS3 Controller Inputs for the serious game.

Following the creation of the action map, the actions need to be initialised in the C# script within the "Awake function" since it is a function used to initialize any variables or game state before the game starts and it is called only once during the lifetime of the script. Considering the script named "Normal Bicycle" that controls the bicycle movement as an example (see Fig. 3.8), it is possible to see how the actions are initialized. First, "FL", "FR" (meaning "force exerted by the left leg", "force exerted by the right leg" respectively) and "brakes", are the names of the variables that are going to receive the output values of the actions inputs. Because the actions "MoveBikeL" and "MoveBikeR" are bound to the PS3 controller thumbsticks these, are going to receive a "Vector2" value since each thumbstick emits two values ranging from -1 to 1, corresponding to the x coordinate and the y coordinate. "Vector2" is a two-value vector. In this context, the y coordinate of "FL" and "FR" is going

to be used as the measurement of the force exerted by the left and right leg respectively, whereas the x component of "FL" is used to steer the bicycle.

An action by itself doesn't represent an actual response to input, an action informs the code that a certain type of input has occurred, and consequently, the code needs to respond to this information [30]. There are several ways to do this, and as such the option chosen was the "started, performed, and cancelled callback" (see Fig. 3.9). Every action has a set of different phases, in response to an input:

- Disabled, meaning the action is disabled and can't receive input,
- Waiting, the action is enabled and is actively waiting for input,
- Started, the "Input System" has received input that started an interaction with the action,
- Performed, an interaction with the action has been completed,
- Canceled, an interaction with the action has been cancelled.

To read the current phase of an action, it is necessary to use the "InputAction.phase" (see Fig. 3.9). Each callback receives an "InputAction.CallbackContext" structure, which holds context information that can be used to query the current state of the action and to read out values from controls that triggered the action (InputAction.CallbackContext.ReadValue) [30].

```
void Awake() //like start but called before start
{
    controls = new Input();

    controls.Gameplay.MoveBikeL.performed += ctx => FL = ctx.ReadValue<Vector2>();
    controls.Gameplay.MoveBikeL.canceled += ctx => FL = Vector2.zero;

    controls.Gameplay.MoveBikeR.performed += ctx => FR = ctx.ReadValue<Vector2>();
    controls.Gameplay.MoveBikeR.canceled += ctx => FR = Vector2.zero;

    //performed  An Interaction with the Action has been completed.
    controls.Gameplay.Brake.performed  += ctx => brakes = ctx.ReadValue<float>();
    //canceled An Interaction with the Action has been canceled.
    controls.Gameplay.Brake.canceled += ctx => brakes = 0;
}
```

Figure 3.8: Initializing the action map in the script.

```
var action = new InputAction();

action.started += ctx => /* Action was started */;
action.performed += ctx => /* Action was performed */;
action.canceled += ctx => /* Action was canceled */;
```

Figure 3.9: Action callback phases - Unity example [30].

3.2 Game modes and physics of motion

3.2.1 The different game modes

Taking into account the different needs of each patient, and having a better ability to integrate said conditions, the serious game for the static bicycle has three different game modes.

The first game mode intends to replicate the behaviour of a standard bicycle, being the only difference not falling to the sides when the user stops pedalling. In order to move, the user must exert force on the pedals continuously; consequently, the more force applied to each pedal, the greater the bicycle velocity will be.

The second game mode is constant velocity, which means the user does not need to exert force on the pedals for the bicycle to move. The bicycle continues to move during the duration of the exercise which makes the pedals move alongside.

The last game mode can compensate for one or both legs in case the user has any mobility issues that compromise the strength of one leg compared to the other or doesn't possess enough strength in either one of the legs to make the pedals move evenly. Besides the compensation, this mode behaves the same way as the first game mode.

Additionally, the user is able to choose the direction in which the bicycle is headed, stop the bicycle movement by pressing the brakes at any given time, and can change the "camera view" to look around the scenario. These characteristics are present in all three game modes. In addition, some variables can be adjusted, for example, there are ten levels of resistance of the pedals that the user can choose according to their individual needs, in all the different game modes. In the particular case of constant velocity, the user can choose the desired velocity of the bicycle for the exercise. The weight of the player also has an impact on the bicycle movement and is registered at the beginning of the game.

3.2.2 Rigid body and colliders

In order to better understand how the bicycle movement was made to replicate a real bicycle, firstly, it is crucial to understand Unity's important classes [33]. One such example is the class "GameObject". This class represents anything which can exist in a "Scene". In turn, a "Scene" is simply the environment in which it is possible to work with content in Unity [34].

Game objects can be characters, props, scenarios, lights etc (see Fig. 3.10). On their own, they do not do anything, consequently, they work as containers for functional components, that control how the game objects look and behave.

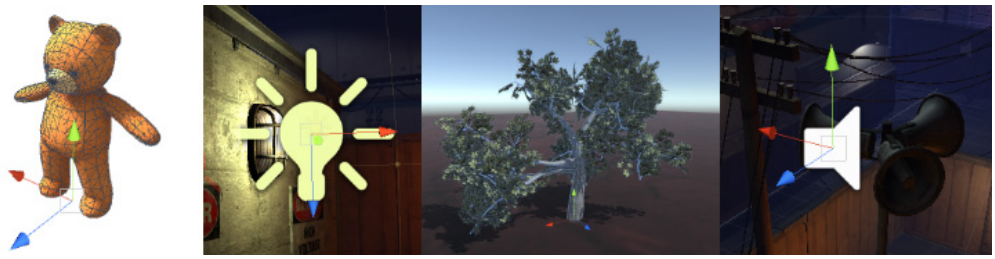


Figure 3.10: Four different types of GameObject: an animated character, a light, a tree, and an audio source [35].

Components exist in order to give a game object the properties it needs to reach a desired outcome. By adding different combinations of components to a game object it is possible to create different types of game objects. For example, if a light component is added to a game object a light source appears in the scene. On the other hand, if the object is a cube the components needed to create it are "Mesh Filter" and "Mesh Renderer" to draw the surface of the cube, then a "Box Collider", so the object can have a volume and in turn physics (see Fig. 3.11). Furthermore, developers can create their own components by writing a script and attaching it to the desired game object.

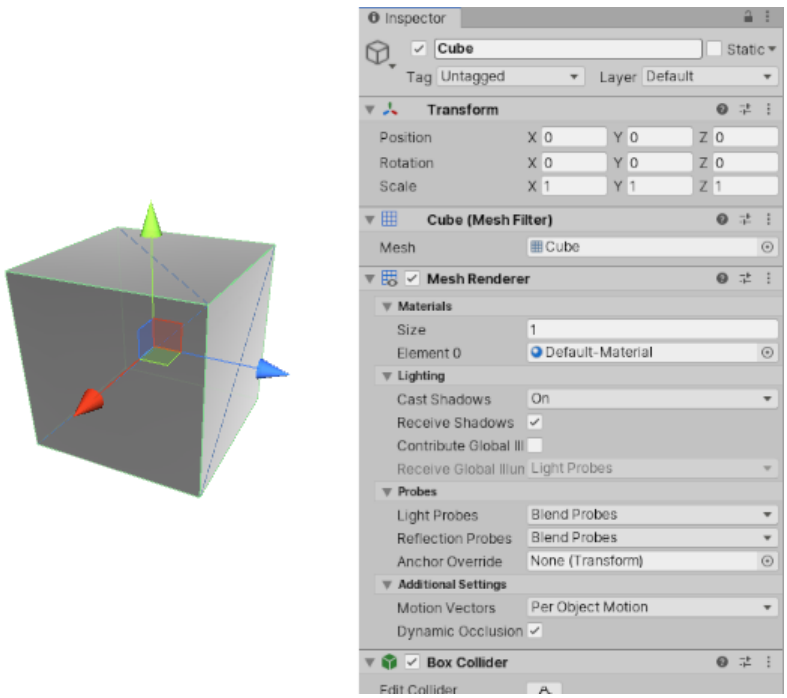


Figure 3.11: Components needed to create a cube, [35].

In the first game mode, an empty game object called "First person player" was created, this game object is the parent of the bicycle model and the main camera. In this way, the movement of the bicycle and the camera are synchronized (see Fig. 3.12). Then, the components "Rigid Body", "Sphere Collider" and a control script called "Normal Bike Mode" were added to the parent object (see Fig. 3.13).

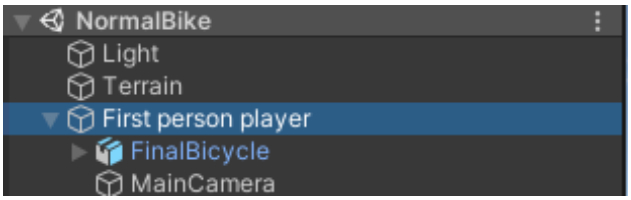


Figure 3.12: Game objects hierarchy.

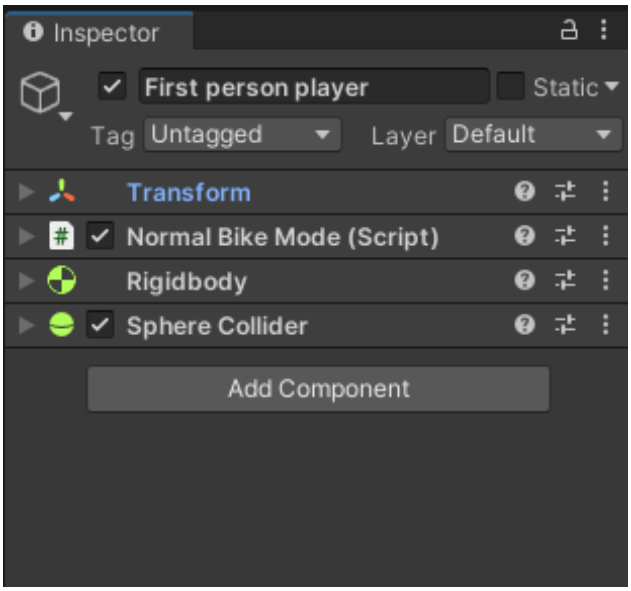


Figure 3.13: Game objects components.

A rigid body component allows the game object to behave according to Unity's built-in physics engine calculations [36]. In this way, it is possible to ensure that objects respond to collisions and accelerate correctly while being impacted by gravity and various other forces. Instead of using the default "Transform" properties (position, rotation and scale of an object) [37], the game object movement can be controlled by simulated physics forces and torque. Additionally, the rigid body component offers a range of properties to the game object such as mass, drag, angular drag, "use gravity", "is kinematic", "interpolate" and many others.

Moreover, it is possible to monitor the rigid body's behaviour, through the properties of speed, velocity, angular velocity, local center of mass, inertia tensor, world center of mass and sleep state. These components contain data about the rigid body and can only be altered

via script although it is not recommended. These values should only be altered by applying forces through the physics system [38].

For the game object to be affected by collisions correctly the physical boundaries of its rigid body component must be defined as well as its mass, [39]. Therefore, it is required to add collider components to the game object, such as wheel colliders and mesh colliders to create the collision response, and a collider associated with the same game object as the rigid-body, so a center of mass can be created.

In this case, a sphere collider was chosen to create the center of mass of the bicycle, which possesses the weight of the player and the bicycle frame combined. This sphere was placed in the coordinates $(0, 1, 0)$, x, y and z respectively, in relation to the bicycle's local axis system. Although the wheels have a set mass of 2 kg each, the mass of the person using the bicycle and its body distribution change for each individual. In addition, it is impossible to determine how each person positions their body on the bicycle, which can alternate multiple times during the course of the exercise. All these factors can alter the center of mass, that's why this set of coordinates was chosen (see Fig. 3.14).

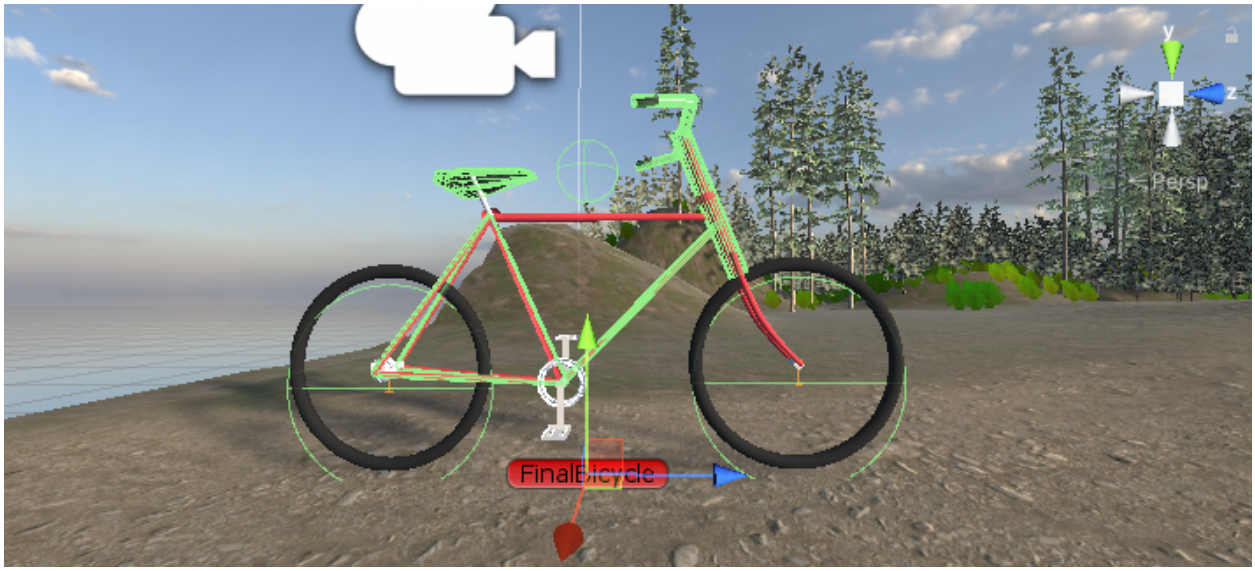


Figure 3.14: Bicycle's sphere collider and bicycle model.

Moreover, a mesh collider component [40] was also added to some parts of the bicycle body (see Fig. 3.15). This, however, does not affect the center of mass in any way, only providing collision detection to the bicycle's frame. Additionally, the convex property was enabled, which allows the mesh collider to collide with other meshes in the environment (see Fig. 3.16).

Figure 3.15: Bicycle frame with collision mesh.

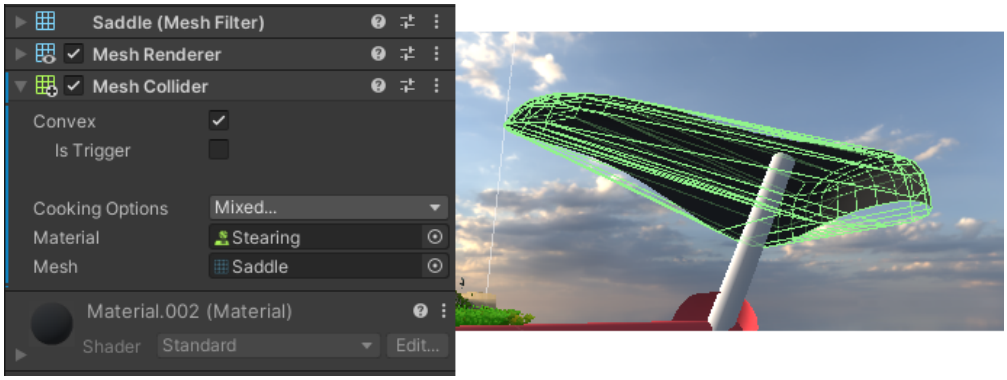


Figure 3.16: Collision mesh component saddle example.

3.2.3 Wheel Collider component

Unity offers a variety of collider components one of which is the "wheel collider". This type of collider is designed specifically for grounded vehicles, as it possesses built-in collision detection, wheel physics, and a slip-based tire friction model. The wheel collider component is powered by "PhysX 3 Vehicles SDK" [41]. This algorithm models vehicles by grouping sprung masses, each one of the masses represents a suspension line with associated wheel and tire information. Furthermore, the complementary representation of these groupings of sprung masses is a rigid-body actor whose mass, center of mass, and moment of inertia precisely match those of the sprung masses and their coordinates (see Fig. 3.17).

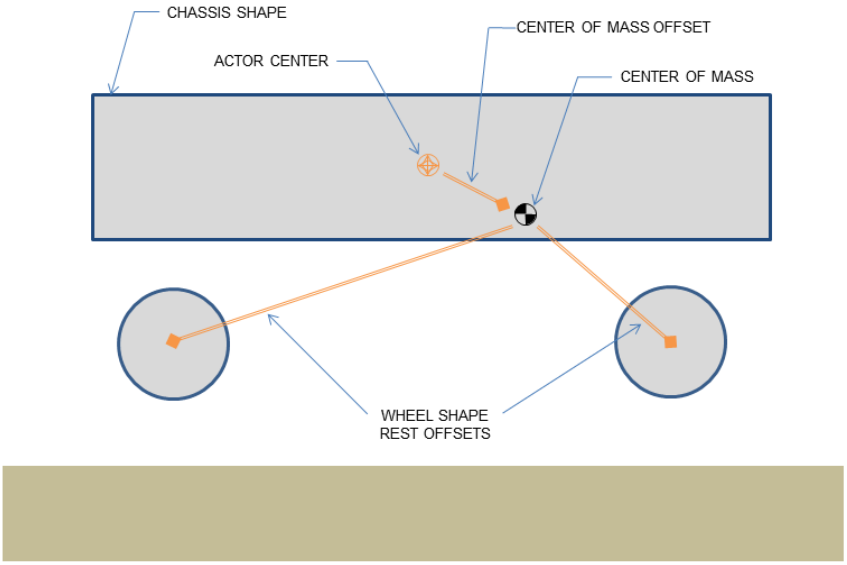


Figure 3.17: Vehicle representation as a rigid body actor with the shapes of the chassis and wheels [41].

The mathematical formula used to calculate the suspension and tire forces is based on the relationship between the rigid-body vehicle representations and sprung masses which can be formalized with the rigid-body center of mass equations:

$$M = M_1 + M_2 \tag{3.1}$$

$$Xcm = (M_1 \cdot X_1 + M_2 \cdot X_2) \,/\, (M_1 + M_2) \tag{3.2}$$

where M is the rigid body mass, Xcm is the rigid body center of mass offset, M_1 and M_2 the sprung masses. X_1 and X_2 are the sprung mass coordinates in actor space (see Fig. 3.18).

The results of these equations are forces that are then applied to the "PhysX SDK" rigid-body representation in the form of a modified velocity and angular velocity. Additionally, the interactions between the rigid-body actor and other objects in the scene as well as its global position are managed by the "PhysX SDK".

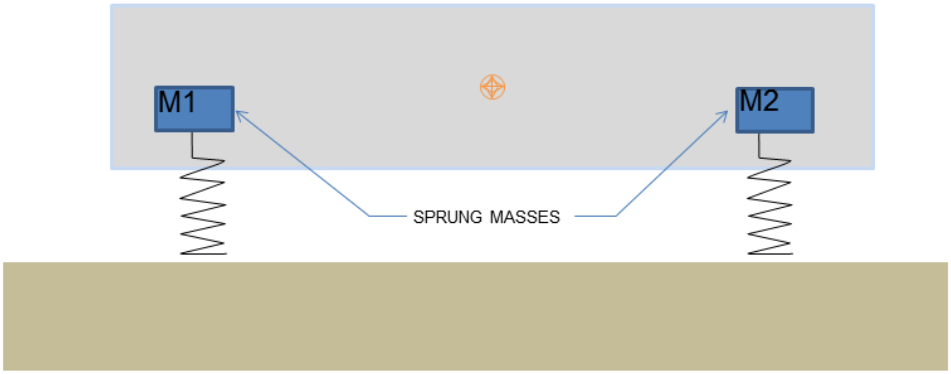


Figure 3.18: Vehicle representation as a group of sprung masses, M_1 and M_2 [41].

The vehicle's update initiates with a raycast of each suspension line. The raycast is positioned at the top of the wheel at maximum compression and cast downwards along the direction of the suspension line to a position slightly below the bottom of the wheel at maximum droop (see Fig. 3.19). The aggregation of the suspensions forces from each compressed or elongated spring is computed and added to the rigid-body actor. Furthermore, the force bearing down on the wheel is computed by the suspension force, which is used to calculate the wheel forces that will be generated in the contact plane and then added to the aggregate force to be applied to the rigid-body actor. Its transform is then modified accordingly in the next "PhysX SDK" update. This computation is dependent on a number of factors such as friction, wheel rotation speed, steering angle, rigid-body momentum and camber angle.

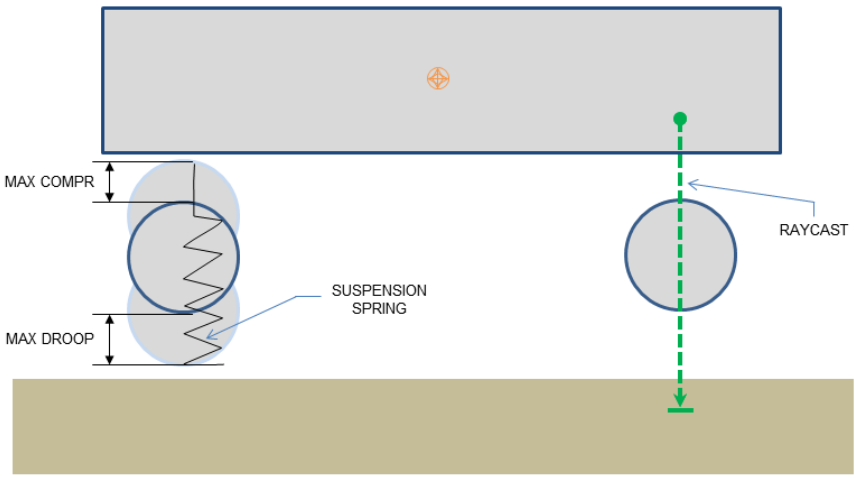


Figure 3.19: Suspension raycasts and limits [41].

Additionally, "PhysX" vehicles are collections of sprung masses and support a wide range

of drive models. The torsion clutch is placed at the center of the driver model, which connects the engine and the wheels by forces resulting from differences in rotational speeds at either side of the clutch. The engine, modelled by a rigid-body with a single degree of rotational freedom and pure rotational motion, is powered directly by the accelerator pedal and is placed on one side of the clutch, whereas, on the other side are wheels, the gearing system and the differential. The wheels are grouped to the clutch through the differential, this allows the calculation of the effective rotational speed of the other side of the clutch, which is computed directly from the gearing ratio and the rotational speed of the wheels. Just like in a typical car, this model naturally enables engine torques to propagate to the wheels and wheel torques to propagate back to the engine.

The wheel collider component possesses its own set of properties [42], some examples are:

- Mass, the mass of the wheel expressed in kilograms,
- Radius, the radius of the wheel, measured in local space,
- Wheel damping rate, the value of damping applied to a wheel,
- Suspension distance describes the maximum extension distance of wheel suspension, measured in local space, the suspension always extends downwards through the local Y-axis as it moves along the local up vector of the rigid body from the coordinate of the wheel center at maximum spring elongation to the coordinate of the wheel center at maximum spring compression. It is expressed in metres,
- Force app point distance, this parameter defines the point where the wheel forces will apply. This is expected to be in metres from the base of the wheel at rest position along the suspension travel direction. When $forceAppPointDistance = 0$ the forces will be applied at the wheelbase at rest. A better vehicle would have forces applied slightly below the vehicle's centre of mass,
- Center, the center of the wheel in object local space,
- Suspension spring, the suspension attempts to reach a Target Position by adding spring and damping forces,
- Spring, the spring force attempts to reach the target position. A larger value makes the suspension reach the target position faster,

- the damper, dampens the suspension velocity, a larger value makes the suspension spring move slower,
- Target Position is the suspension's rest distance along suspension distance that sets the rest coordinate of the wheel to the mid-point between the suspension at maximum elongation and maximum compression,
- Forward and sideways friction, properties of tire friction when the wheel rolls forward and sideways.

Since the wheel collider component is mainly targeted for the simulation of cars, some alterations were done to the default values (see Fig. 3.20).

First, the mass of each wheel was changed from 20 kg to 2 kg since the average weight of a bicycle wheel ranges from 1 kg to 2 kg [43]. The radius was adapted to fit the size of the bicycle model tires, 0.35 m. These were designed to have a 0.7 m diameter, the standard size of bicycle wheels. Then the suspension distance was shortened to 0.08 m, within the standard values range since bicycles have a shorter suspension distance compared to cars. The wheel dampening rate was kept at 0.25, and the force app point distance of the wheels changed to 0.35 m, the center, to avoid undesired behaviours such as jittering, moving along the surface without input, or falling [44]. The suspension spring values were altered as well since the weight of the bicycle can never reach the weight of a car frame. The spring value was changed from 35000 N·m to 35 N·m. Usually, the first value corresponds to a 1500 kg car body, while the bicycle plus the player weight only reaches a maximum of 150 kg. The damper value was set at 4.5 N·s and the target position was set at 0.1 because 0.5 is the default value which matches the behaviour of a regular car's suspension.

Furthermore, the forward and sideways friction values were maintained (see Fig. 3.21) in order to create an adequate wheel friction curve. It is possible to see the "Extremum Slip/Value" which is the curve's "extremum" point, "Asymptote Slip/Value" is the curve's "asymptote" point and stiffness which corresponds to the multiplier for the "extremum" value and "asymptote" value which changes the stiffness of the friction. If the stiffness value is set to zero the friction from the wheel will be completely disabled, for that reason the default value is one.

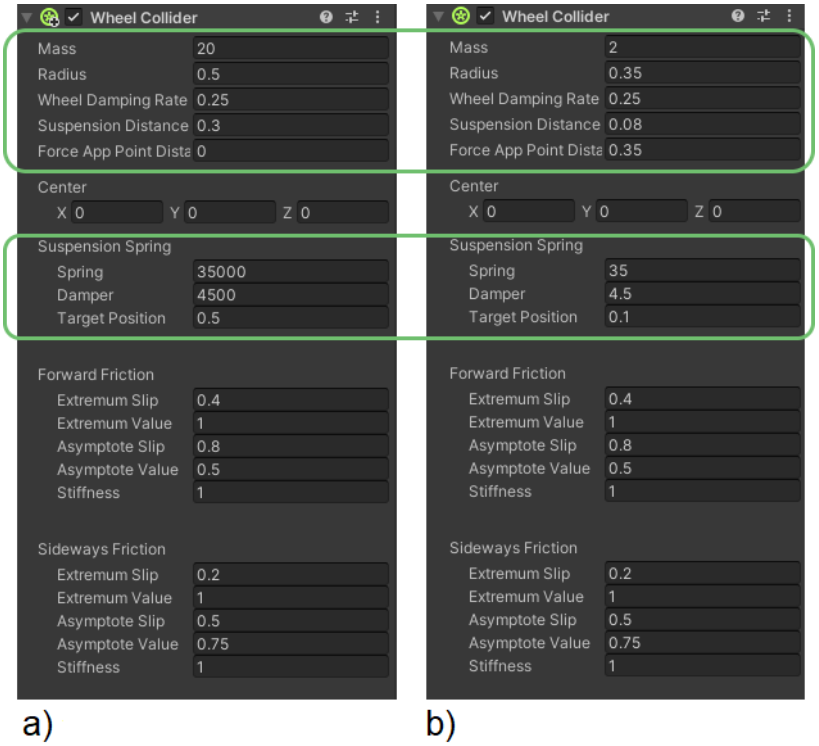


Figure 3.20: a) Wheel collider component default values. b) Wheel collider component altered values.

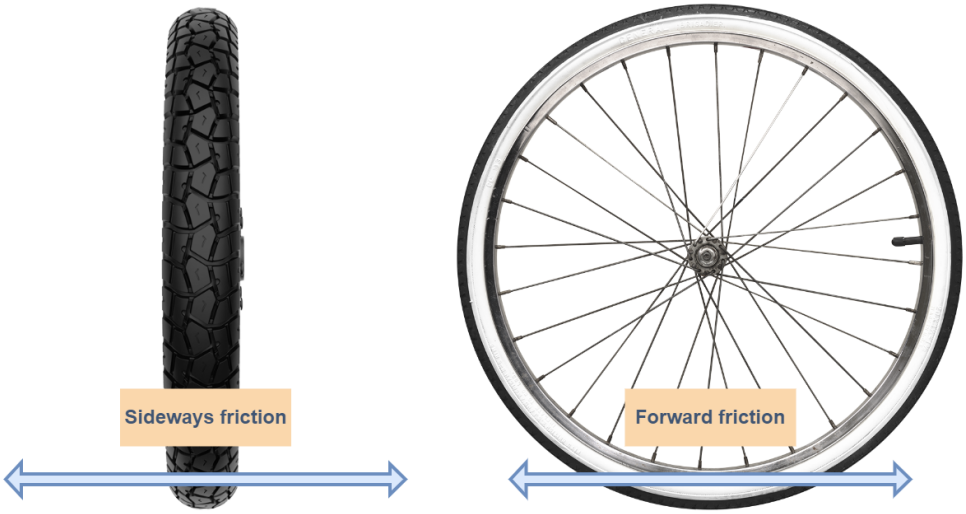


Figure 3.21: Wheel sideways and forward friction [42].

The wheel friction plot takes as an input value of tire slip and returns a force as output. Before reaching the extremum point, the slipping of the tire is negligible; after that point slipping occurs, too much is not desirable and can make the bicycle fall or not be able to

stop when need be, hence the need to keep these values balanced (see Fig. 3.22).

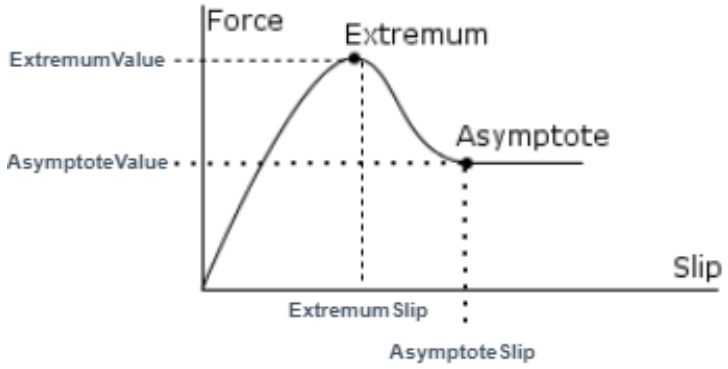


Figure 3.22: Wheel friction curve [42].

3.2.4 Bicycle control scripts

The wheel colliders do not roll or turn. The objects that have the wheel collider attached should permanently be fixed relative to the vehicle itself. Nevertheless, it is possible to create a graphical representation of the wheels turning and rolling [42].

In order to achieve this result the wheel meshes and the wheel colliders were set up in different game objects. Additionally, the creation of a function to update the wheel mesh's transform was required (see Fig. 3.24). This function was named "UpdateWheel", and receives as inputs the wheel collider component as well as the wheel mesh transform (see Fig. 3.23).

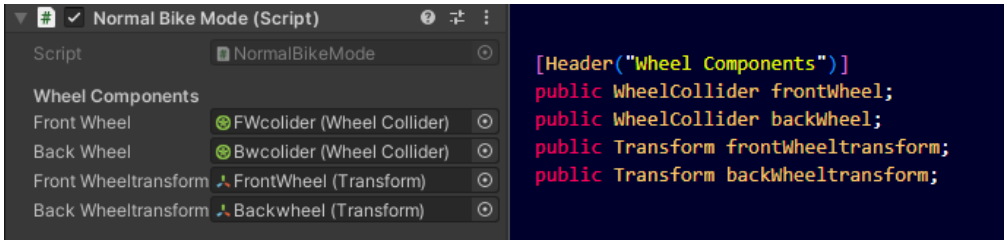


Figure 3.23: Wheel components in the inspector (left) and declared in the scrip, (right).

The public method "GetWorldPose", obtains the world space position of the wheel collider, in the form of a 3D vector, and rotation angle in the form of a quaternion, whilst considering the steering angle, ground contact and suspension limits [40]. Afterwards, this data is transferred to the wheel mesh's transform, which will acquire these values for position and rotation, since this function updates one wheel mesh at a time, it was called twice (see Fig. 3.24).

```
void UpdateWheel(WheelCollider wheelCollider, Transform wheelTransform)
{

    wheelCollider.GetWorldPose(out wheelpos, out wheelrot);

    wheelTransform.position = wheelpos;
    wheelTransform.rotation = wheelrot;
}
```

Figure 3.24: Function required to update the wheel mesh's position and rotation.

For the player to be able to steer the bicycle a function named "Steer" was created. This function receives as inputs the horizontal component of "FL" as mentioned previously on 3.1, "maxangle", which is the maximum angle in degrees that the bicycle can rotate around the local y-axis, and "smoothing", the interpolation value which can range from 0 to 1 and set at 0.5.

To calculate the new angle of steering an interpolation function "Mathf.LerpAngle" [45], which calculates the linear interpolation between two angles is used. The first parameter is the current steering angle value, and the second parameter is the product between the maximum steering angle and the horizontal component of "FL". This interpolation will be done by a smoothing value of 0.5, which is the third parameter. Afterwards, the updated steering angle will be applied to the front wheel's steering angle (see Fig. 3.25).

```
[Header("Wheel Variables")]
6 references
public float currentSteeringAngle = 0;
2 references
public float targetlayingAngle = 0;
2 references
float maxangle = 15f;
3 references
float smoothing = 0.5f;

void FixedUpdate()
{


    horizontal = FL.x;

1 reference
public void Steer(float maxangle , float smoothing, float horizontal)
{   //Steering angle in degrees, always around the local y-axis.

    currentSteeringAngle = Mathf.LerpAngle(currentSteeringAngle, maxangle * horizontal , smoothing);
    frontWheel.steerAngle = currentSteeringAngle;
}
```

Figure 3.25: Function required to control the steering of the bicycle.

When the bicycle changes its course, not only the front wheel has to turn but also

the bicycle's handlebar and its body. For that, a function named "UpdateHandlebar" was created. This function will update the local rotation of the handlebar by a quaternion function, "Quaternion.Euler", which returns a 3D vector with x degree rotation around the x axis, for the first coordinate, a y degree rotation around the y axis for the second coordinate and lastly, a z degree rotation around the z axis for the third coordinate [46]. In this particular case, the only rotation needed is the one around the y axis. As such, the first and third coordinates were maintained, whilst the second coordinate is updated by the steering angle, calculated by the previous function.

```
public void UpdateHandlebar()
{
    handlebar.localRotation = Quaternion.Euler(handlebar.localRotation.eulerAngles.x, currentSteeringAngle,
    handlebar.localRotation.eulerAngles.z);
}
```

Figure 3.26: Function required to update the handlebar rotation.

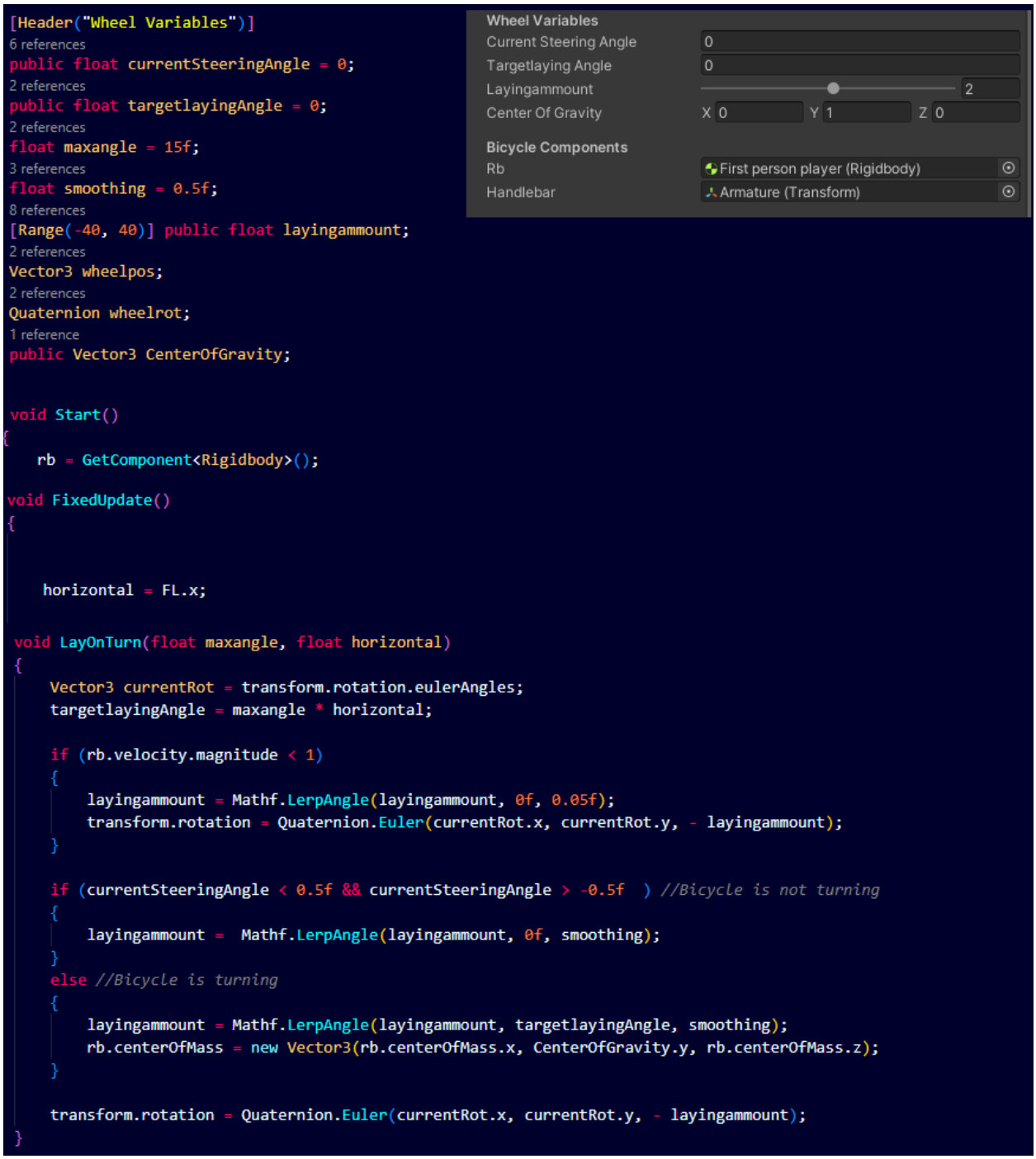
Since the wheel collider component is targeted mainly for four-wheel vehicles an additional function was needed to maintain the equilibrium of the bicycle. This was achieved by checking the values of the steering angle to confirm if the bicycle is straight or turning and controlling the amount of rotation around the z-axis. First, a 3D vector, which contains the rotation of the first-person player (see section 3.2.2) in degrees was created, then the amount of laying done by the player, "targetlayingAngle", is calculated by the product of the maximum steering angle and the horizontal input.

If the first-person player velocity magnitude is less than 1, this indicates the bicycle is close to being still, consequently, the laying amount will be calculated with the function "Mathf.LerpAngle" [45], which will be an interpolation between the laying amount and 0. Afterwards, the rotation in degrees of the first-person player around the z-axis will be updated to the laying amount previously calculated.

If the current steering angle, formerly calculated by the function "Steer", ranges between -0.5 and 0.5, means the bicycle is not turning. As such, the rotation of the player will not be updated. Nevertheless, the laying amount will still be calculated as it was previously.

Finally, when turning occurs, the new value for the laying amount will be calculated by "Mathf.LerpAngle", an interpolation between the laying amount and "targetlayingAngle", then the center of mass of the rigid body will be altered in the y-axis by the value stored in a 3D vector called "CenterOfGravity" that contains the coordinates of the sphere collider, which acts as the center of mass of the first-person player. Furthermore, the rotation in

degrees of the first-person player around the z-axis will be updated to the laying amount calculated (see Fig. 3.27).




```
[Header("Wheel Variables")]
6 references
public float currentSteeringAngle = 0;
2 references
public float targetlayingAngle = 0;
2 references
float maxangle = 15f;
3 references
float smoothing = 0.5f;
8 references
[Range(-40, 40)] public float layingammount;
2 references
Vector3 wheelpos;
2 references
Quaternion wheelrot;
1 reference
public Vector3 CenterOfGravity;


void Start()
{
    rb = GetComponent<Rigidbody>();

void FixedUpdate()
{


    horizontal = FL.x;

 void LayOnTurn(float maxangle, float horizontal)
 {
     Vector3 currentRot = transform.rotation.eulerAngles;
     targetlayingAngle = maxangle * horizontal;

     if (rb.velocity.magnitude < 1)
     {
         layingammount = Mathf.LerpAngle(layingammount, 0f, 0.05f);
         transform.rotation = Quaternion.Euler(currentRot.x, currentRot.y, - layingammount);
     }

     if (currentSteeringAngle < 0.5f && currentSteeringAngle > -0.5f  ) //Bicycle is not turning
     {
         layingammount =  Mathf.LerpAngle(layingammount, 0f, smoothing);
     }
     else //Bicycle is turning
     {
         layingammount = Mathf.LerpAngle(layingammount, targetlayingAngle, smoothing);
         rb.centerOfMass = new Vector3(rb.centerOfMass.x, CenterOfGravity.y, rb.centerOfMass.z);
     }

     transform.rotation = Quaternion.Euler(currentRot.x, currentRot.y, - layingammount);
 }
```

Figure 3.27: Function required to control the bicycle's balance.

For the bicycle to accelerate and brake a function named "Accelerateandbrake" was added. This function will take as input the force exerted by the player on the bicycle pedals, a variable named "TotalForce".

The wheel collider property "motorTorque" [42], allows a simulation of torque on the wheels of a vehicle in newtons. As such, the "TotalForce" was used applied to the back

wheel of the bicycle via this property. In addition, a property named "brakeTorque" allows for the simulation of brakes in newtons. Consequently, an if statement was used to check if the brake button was being pressed or not (see section 3.1). If pressed, the brakes of both wheels are given the double of the player's mass to stop the bicycle's movement, otherwise, this value will be 0 (see Fig. 3.28).

```
public void Accelerateandbrake(float TotalForce)
{

    backWheel.motorTorque = TotalForce;
    brakeForce = rb.mass * 2;


    if (brakes > 0)
    {
        frontWheel.brakeTorque = brakeForce;
        backWheel.brakeTorque =  brakeForce;
    }
    else
    {
        frontWheel.brakeTorque = 0;
        backWheel.brakeTorque = 0;
    }

}
```

Figure 3.28: Function required to accelerate and brake.

Since there are three different game modes, the variable that makes the calculations for the "TotalForce" are different in each one. For example, in the normal bicycle mode, the force will be the average of the forces exerted by both legs, multiplied by the rigid body's mass, the conjunction of the players and the bicycle's frame masses, which in turn will be multiplied by a factor of 0.1 and gravitational acceleration. The factor was chosen due to the fact the values received by the inputs vary between -1 and 1 (see Fig. 3.29). The calculation of the average force of each leg was chosen, since the legs are out of phase from each other, and only exert force for each half-cycle.

```
TotalForce = ( (FR.y/2) + (FL.y/2) ) * (rb.mass * 9.8) * 0.1; // Newtons 10% do peso da pessoa nos pedais
horizontal = FL.x;
```

Figure 3.29: Calculation of the force exerted in normal bicycle mode.

In the second mode, leg compensation, an if statement is used to check which leg is in need of compensation. The total force calculations will occur as follows for the right and left leg respectively:

$$Totalforce = \begin{cases} \left(\frac{FR}{2} + \left(\frac{FL}{2} \cdot Comp\right)\right) + \frac{FL}{2} & \text{if right leg is in compensation} \\ \left(\frac{FL}{2} + \left(\frac{FR}{2} \cdot Comp\right)\right) + \frac{FR}{2} & \text{if left leg is in compensation} \end{cases} \tag{3.3}$$

where $Totalforce$ is the sum of the force exerted on both pedals, FR is the force exerted by the right leg, FL is the force exerted by the left leg and $Comp$ is the amount of compensation needed.

In this way, a percentage of the force exerted by the healthy leg is used as compensation for the damaged one, similarly to the previous mode the force value will also be multiplied by the rigid body mass, gravitational acceleration and a factor of 0.1 (see Fig. 3.30).

```
if ( PlayerDataSO.LegLeft == true){TotalForce = (  ( (FL.y/2) + ((FR.y/2) * (PlayerDataSO.Legcompensation/100)) ) + (FR.y/2)  )
* 9.8 * rb.mass * 0.1;}
else if( PlayerDataSO.LegRight == true){TotalForce = (  ( (FR.y/2) + ((FL.y/2) * (PlayerDataSO.Legcompensation/100)) ) + (FL.y/2)  )
* 9.8 * rb.mass * 0.1;}
```

Figure 3.30: Calculation of the force exerted in leg compensation mode.

Finally, for the constant velocity mode, the values of the leg inputs are not taken into account, since the goal of this mode is for the bicycle to move at constant velocity independent of the force exerted on the pedals. Consequently, the velocity is decided at the beginning of the game and the player can only steer the bicycle, brake, look around, and pause. To simulate a constant velocity a torque will be applied to the "motorTorque" and the "brakeTorque", just like previously (see Fig. 3.31).

```
public void Accelerateandbreak(float torque)
{

    backWheel.motorTorque = torque;
    currentbrakeForce = torque * 2;// the breakforce will be double the force of the pedals


    if (breaks > 0)
    {
        frontWheel.brakeTorque = currentbrakeForce;
        backWheel.brakeTorque =  currentbrakeForce;
    }
    else
    {
        frontWheel.brakeTorque = 0;
        backWheel.brakeTorque = 0;
    }

}
```

Figure 3.31: Function required to accelerate and brake, constant velocity mode.

Independently from the game modes, the player can always look around the scene for a more immersive experience to simulate this, a script component named "Player view" was added to the main camera (see Fig. 3.32).

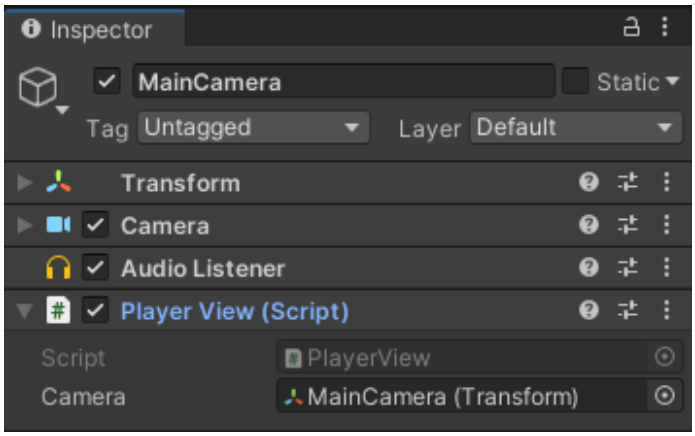


Figure 3.32: Main camera and respective components.

This script uses "Xleft", "Xright", "Yup" and "Ydown" as inputs, (see section 3.1), (see Fig. 3.33) and creates values for the maximum rotation around the x and y axis. Since Euler angles only range from 0 to 360 degrees, the values establish for the rotation around the x-axis were 25 degrees upwards and 335 degrees downwards. For the rotation around the y-axis, the values were 50 degrees to the right and 310 degrees to the left (see Fig. 3.34).

```
controls = new Input(); //create a input object

controls.Gameplay.Xleft.performed += ctx => xleft = ctx.ReadValue<float>(); //performed  An Interaction with the Action
controls.Gameplay.Xleft.canceled += ctx => xleft = 0;//canceled An Interaction with the Action has been canceled.

controls.Gameplay.Xright.performed += ctx => xright = ctx.ReadValue<float>();
controls.Gameplay.Xright.canceled += ctx => xright = 0;

controls.Gameplay.Yup.performed += ctx => yup = ctx.ReadValue<float>();
controls.Gameplay.Yup.canceled += ctx => yup = 0;

controls.Gameplay.Ydown.performed += ctx => ydown = ctx.ReadValue<float>();
controls.Gameplay.Ydown.canceled += ctx => ydown = 0;
```

Figure 3.33: Initialization of the action map in the "Player view" script.

```
2 references
float maxrotR = 50f; //the amount the player can turn to the right
2 references
float maxrotL = 310f; //the amount the player can turn to the left 310 = 360 - 50

2 references
float maxrotU =25f; //the amount the player can turn upwards
2 references
float maxrotD = 335f; //the amount the player can turn downwards 335 =360 - 25
```

Figure 3.34: Range of values established for the rotation around the x-axis and y-axis.

Afterwards, the local rotation of the camera in degrees around the x and y axis was stored in "xRotation" and "yRotation" respectively. Additionally, an if statement was used to check if the amount of rotation is within the limits previously established. If the values were indeed within the y-axis range of values and "xleft" is pressed, then "Vector3 down", a shorthand for writing $Vector3(0, -1, 0)$ is used and, consequently, the camera rotates to the left. On the other hand, if "xright" is pressed "Vector3.up", a shorthand for writing $Vector3(0, 1, 0)$, is used, making the camera rotate to the right. This process is repeated for the rotation around the x-axis, with the range of values mentioned previously (see Fig. 3.35).

```
if (yRotation <= maxrotR  || yRotation >= maxrotL){
    //se a camara estiver entre os angulos pretendidos ocorre rotação

    if( xleft > 0 ){ //verefies if the button was pressed

        //Transform.Rotate to rotate GameObjects.
        // The rotation is often provided as an Euler angle and not a Quaternion.
        transform.Rotate(Vector3.down);//down Shorthand for writing Vector3(0, -1, 0)
       //represents the axis arround wich to turn (-y)
       //its going to move by 1 each frame because its inside update

     }

    if( xright > 0 ){

        transform.Rotate(Vector3.up);//up shorthand for writing Vector3(0,1,0)

    }
}
if (xRotation <= maxrotU || xRotation >= maxrotD){

   if(yup > 0){

       transform.Rotate(Vector3.left);//left shorthand for writing Vector3(-1,0,0)

    }

    if(ydown > 0){

       transform.Rotate(Vector3.right);//right shorthand for writing Vector3(1,0,0)

    }
}
```

Figure 3.35: Conditions used to occur the rotation around the y and x-axis respectively.

The player can press the button until reaching the maximum amount of rotation. To avoid going outside the range defined other conditions were added. Taking the rotation around the y-axis as an example, if the value stored in "yRotation" ranges from 50 to 180 degrees. Then a "Vector3 down" is used to counteract the movement, making it impossible to go past the desired range of values.

Also if the value stored in "yRotation" is within the range 180 to 310 degrees, "Vector3

up" is used to go in the opposite direction of the movement and, consequently, stop the rotation. These conditions were repeated for the rotation around the x-axis as well (see Fig. 3.36).

```
else if ( yRotation > maxrotR && yRotation <= 180f ){ //valores proibidos positivos (direita)

    transform.Rotate(Vector3.down); //corrige desvio do angulo (roda ao contrario)

}
else if (yRotation < maxrotL && yRotation >= 180f){//valores proibidos negativos (esquerda)
    transform.Rotate(Vector3.up); //corrige desvio do angulo (roda ao contrario)
}
else if (xRotation > maxrotU && xRotation < 180f){
    transform.Rotate(Vector3.left);
}
else if (xRotation < maxrotD && xRotation > 180f){
    transform.Rotate(Vector3.right);
}
```

Figure 3.36: Conditions used to control the amount of rotation around the y and x-axis respectively.

3.2.5 Game menus and data management

For the game to function correctly each mode has to be contained in different game scenes [47], so it is possible to work with different variables, scenarios and goals. Furthermore, this asset makes it possible to have game menus and settings.

When working with different scenes, there's a need to save the player's data when switching between them, for example, changing from the main menu to one of the game modes. One way of achieving this is by using scriptable objects, which are data containers that store large amounts of data, independently of class instances. All "prefabs" are able to access the scriptable object's data, avoiding duplicated values which reduces the project's memory usage [48].

In this project, a scriptable object class was created to save player data such as weight, height, game resistance, the percentage of leg compensation, constant velocity value, and which of the legs is in need of compensation, left or right (see Fig. 3.37). Then a scriptable object asset named "Player Data SO" was generated from the previous script (see Fig. 3.38).

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu]
public class PlayerData : ScriptableObject
{
    [SerializeField]   private float _weight;
    [SerializeField]   private float _height;
    [SerializeField]   private float _resistance;
    [SerializeField]   private float _legcomp;
    [SerializeField]   private float _velocity;
    [SerializeField]   private bool _legleft;
    [SerializeField]   private bool _legright;

    public float Weight
    {
        get{ return _weight; }
        set{ _weight = value; }

    }

    public float Height
    {
        get{ return _height; }
        set{ _height = value; }

    }
```

Figure 3.37: Scriptable object class script.

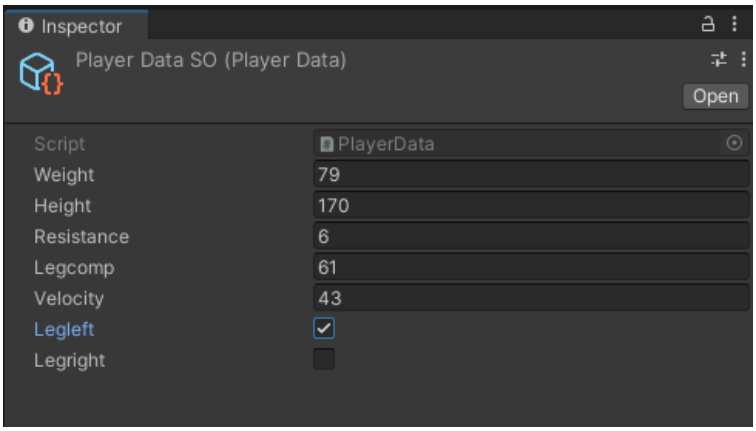


Figure 3.38: Scriptable object asset.

To access the data from the scriptable object through the script, firstly, the creation of a scriptable object variable is needed, in this case, named "Player Data SO", afterwards, it is possible to attribute the data values to the desired game variables. For example, the data containing the weight value was attributed to the rigid body mass, and the data containing the value for resistance was attributed to the rigid body's drag, which simulates the decay

rate of a rigid body's linear velocity [38]. The base value for resistance was set a 0.5, and according to the behaviour of the game, the incrementation had to be done in decimal values, making the resistance range from 0.5 to 1.5, as such, the resistance levels were divided by ten (see Fig. 3.39).

```
[Header("Scriptable Object ref")]
public PlayerData PlayerDataSO;

void Start()//função chamada automaticamente pelo unity no inicio do jogo
{
    rb = GetComponent<Rigidbody>();
    Terrain = GetComponent<Collider>();
    rb.mass = PlayerDataSO.Weight; //changes the rigidbody mass to players weight
    rb.drag = 0.5f + PlayerDataSO.Resistance/10; //changes the drag of the rigid body
}
```

Figure 3.39: Scriptable object declaration and attributions, with weight and resistance examples.

In order to obtain the data values for the scriptable object to store, game menus were created, where the players have the ability to choose the desired game mode and adapt it according to their needs.

For this, the Unity UI package was used [32]. This package allows for the creation of canvas buttons and many other interfaces that compose a game menu (see Fig. 3.40).

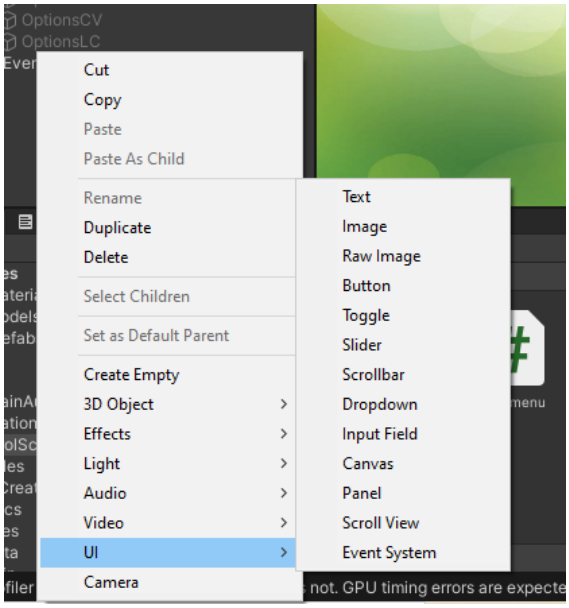


Figure 3.40: Unity UI options.

The main menu is composed of the tile of the game, play and quit buttons. The button component possesses the "onclick" event which allows the activation of scripts or game

objects. This makes it possible to "jump" between different menus on the same scene (see Fig. 3.41).

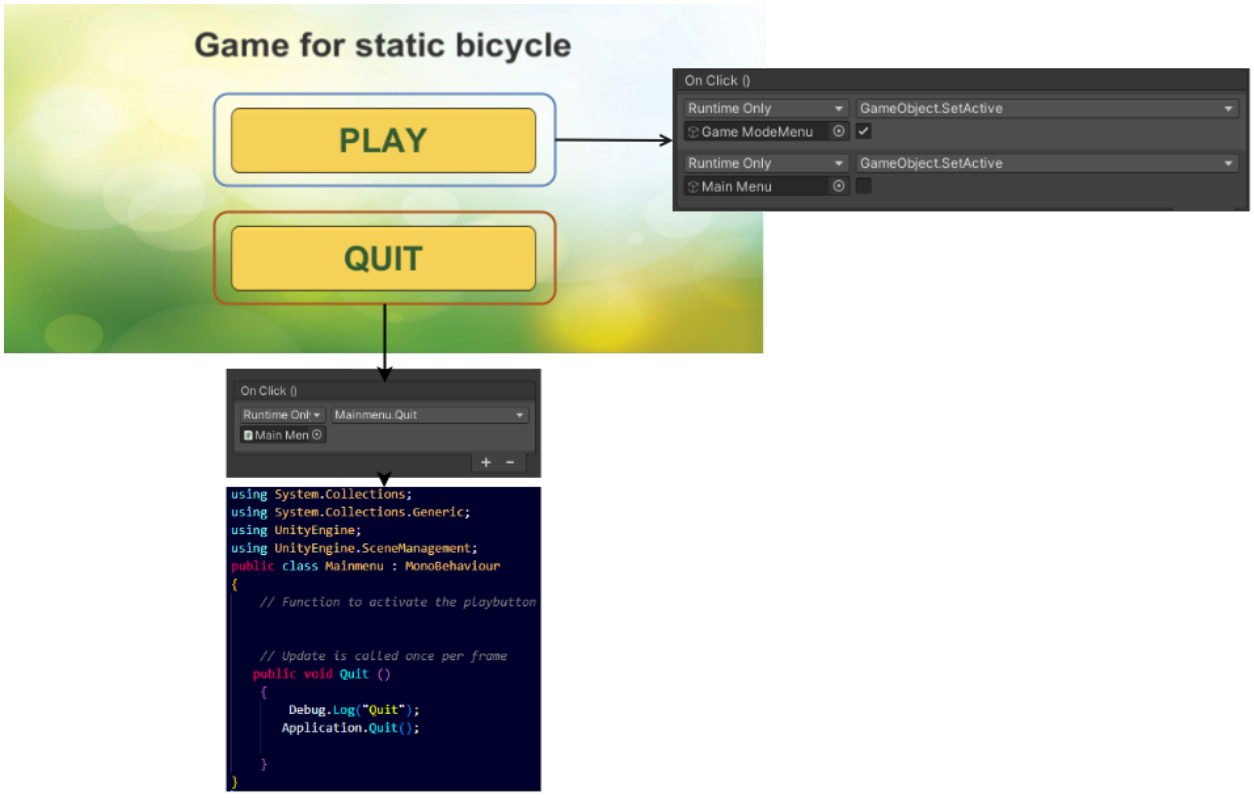


Figure 3.41: Main menu and respective "on click" events.

When the play button is pressed the player will be redirected to the game mode menu. There, it is possible to choose the desired game mode and adjust the options accordingly (see Fig. 3.42).

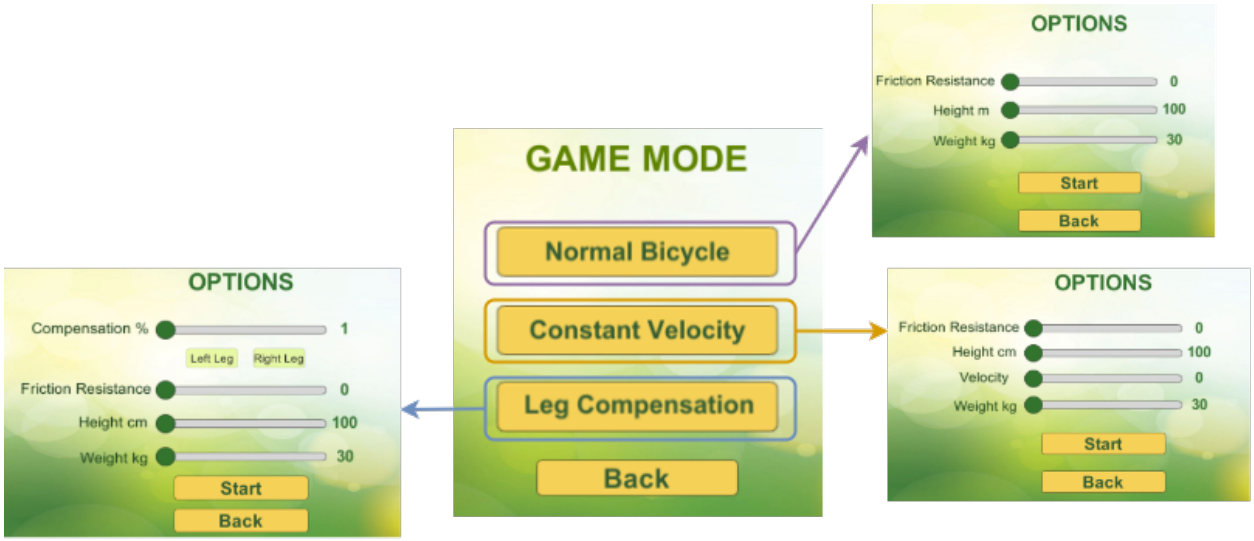


Figure 3.42: Game mode menu and options menus.

Each options menu has slider components. These have minimum and maximum values

attributed to each, to be able to display the value of the slider changing as it is used, a script that gets the value of the slider and updates the slider's text with the value in the form of a string was created. This script is used in all the menu sliders (see Fig. 3.43).

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class slidervalueW : MonoBehaviour
{
    private Slider slider;
    private Text textcomp;

    void Awake()
    {
        slider = GetComponentInParent<Slider>();
        textcomp = GetComponentInParent<Text>();
    }
    void Start()
    {
        UpdateText(slider.value);
        slider.onValueChanged.AddListener(UpdateText);

    }

    void UpdateText(float value)
    {
        textcomp.text = slider.value.ToString();

    }

}
```

Figure 3.43: Script for updating the slider text.

Furthermore, the slider component allows for the usage of "on value changed" event. This makes it possible to store the value of the slider into its scriptable object corresponding variable (see Fig. 3.44).

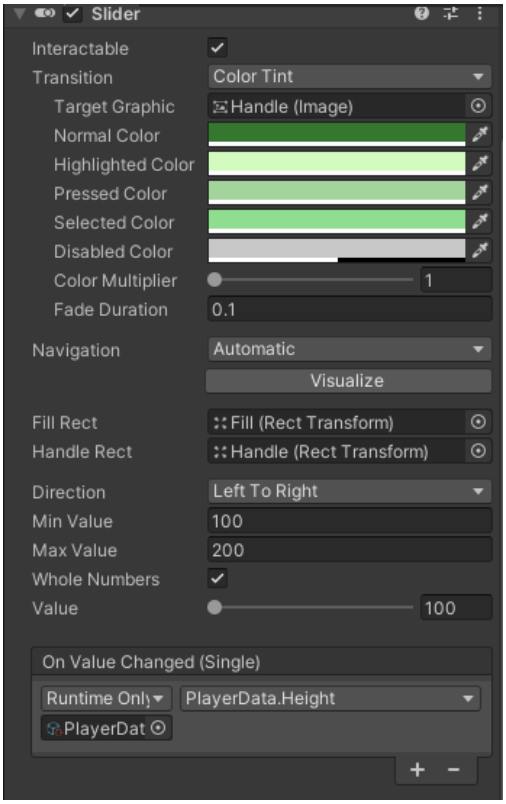


Figure 3.44: Slider component with "on value changed" event for the scriptable object variable height.

Additionally, during the game, the player can pause the exercise and alter the resistance level, at any given time. A timer is also available to monitor the duration of the exercise (see Fig. 3.45).

Figure 3.45: Pause menu and timer.

4 Experimental Validation

Several experiments were carried out in order to:

- Demonstrate that the serious game is user-friendly for a wide range of users,
- Demonstrate that the serious game functions correctly according to the physics engine.

4.1 Usability Questionnaires

In order to evaluate the usability of the serious game, and to have a better perspective for future improvements and developments, usability questionnaires were carried out.

This method of evaluation was chosen since this type of survey allows for the collection of large amounts of data at a relatively low cost.

In favour of selecting the most favourable questionnaire for this work, extensive research on twenty-four different usability questionnaires was done [49]. The usability questionnaires analyzed were the following:

- Questionnaire for User Interface Satisfaction (QUIS),
- Technology Acceptance Model questionnaire (TAM),
- After-Scenario Questionnaire (ASQ),
- Computer System Usability Questionnaire (CSUQ),
- Post-Study System Usability Questionnaire (PSSUQ),
- Software Usability Measurement Inventory (SUMI),
- System Usability Scale (SUS),
- Purdue Usability Testing Questionnaire (PUTQ),
- Website Analysis Measurement Inventory (WAMMI),

- Usefulness, Satisfaction and Ease of use (USE),
- Expectation Ratings (ER),
- Website Usability Evaluation tool (WEBUSE),
- Usability Magnitude Estimation (UME),
- Mobile Phone Usability Questionnaire (MPUQ),
- Single Ease Question (SEQ),
- Website Evaluation Questionnaire (WEQ),
- Subjective Mental Effort Question (SMEQ),
- Usability Metric for User Experience (UMUX),
- Standardized Universal Percentile Rank Questionnaire (SUPR-Q),
- Design-oriented Evaluation of Perceived usability (DEEP),
- Turkish-Computer System Usability Questionnaire (T-CSUQ),
- Usability Metric for User Experience-LITE (UMUX-LITE),
- Speech User Interface Service Quality questionnaire (SUISQ),
- Alternate Usability (AltUsability).

Firstly, the type of user interface or system the questionnaire is meant to evaluate was taken into account. This led to the exclusion of the surveys targeted for mobile phone applications, websites, information-intensive web systems interactive voice response applications, websites of the governmental organizations and information systems.

Then the level of reliability was taken into account. As well most have "very high levels of reliability ranging between 0.9 and 0.97" [49], although, some present levels lower than 0.90 while others don't have precise Cronbach alpha values. Consequently, these were excluded. Analysing the questionnaires left, even though "CSUQ" and "ASQ" have high levels of reliability, 0.95 and 0.96 respectively, they have a limited amount of items compared to the other surveys which, in turn, can make them unreliable. Furthermore, another important factor is the category of the questionnaire. These consist of post-study questionnaires, used at the end of a study, post-task questionnaires for evaluation used at the end of each task or

scenario in a usability study and questionnaires developed solely to evaluate web applications. For the context of this work, the category most adequate is the post-study questionnaires which consist of "SUMI", "PSSUQ" and "SUS". The questionnaire "SUS" (see Table.4.1) is a fast and common method of evaluating software, being the fastest questionnaire to converge on the correct conclusion producing reliable results across multiple sample sizes, frequently used by a large number of usability studies, and referenced in over 600 publications [49]. As a consequence, this was the questionnaire chosen to evaluate this serious game.

Table 4.1: Example of the SUS questionnaire.

	The System Usability Scale Standard Version	Strongly Disagree					Strongly Agree
			1	2	3	4	5
1	I think that I would like to use this system frequently.		o	o	o	o	o
2	I found the system unnecessarily complex.		o	o	o	o	o
3	I thought the system was easy to use.		o	o	o	o	o
4	I think that I would need the support of a technical person to be able to use this system.		o	o	o	o	o
5	I found the various functions in this system were well integrated.		o	o	o	o	o
6	I thought there was too much inconsistency in this system.		o	o	o	o	o
7	I would imagine that most people would learn to use this system very quickly.		o	o	o	o	o
8	I found the system very awkward to use.		o	o	o	o	o
9	I felt very confident using the system.		o	o	o	o	o
10	I needed to learn a lot of things before I could get going with this system.		o	o	o	o	o

In order to evaluate the responses of the participants, the SUS score must be calculated [50]. The formula used to calculate the score is as follows: Firstly the odd-numbered questions generate a positive response, while even-numbered questions generate a negative response, thus their score value has to be inverted. Then answers are recorded with values ranging from one to four, and the resulting points of the ten questions are added, which can result in a maximum value of forty. To create a scale up to one hundred, the user's points are multiplied by 2.5. Finally, this procedure is repeated for all the user's scores which are then averaged. Resulting in the following mathematical expression:

$$SUS = \left(\lim_{n \to 10} \sum (x_{2n-1} - 1) + \lim_{n \to 10} \sum (5 - x_{2n}) \right) \cdot 2.5 \tag{4.1}$$

where n is the number of the question, $2n$ represents the even questions, $2n-1$ represents the odd questions, and x is the score of the question (see Table 4.2).

Table 4.2: Results of the SUS questionnaire.

	Question	1	2	3	4	5	6	7	8	9	10	SUS score
User												
1		4	3	4	2	4	2	4	3	4	3	67.5
2		4	1	5	2	4	2	4	2	4	2	77.5
3		3	1	4	2	5	1	4	2	4	2	80
4		5	2	2	2	4	1	5	1	4	2	80
5		4	2	5	2	4	2	4	3	5	2	77.5
6		5	2	5	2	5	2	5	1	5	1	92.5
7		4	2	3	1	2	2	4	3	3	1	67.5
8		3	1	5	2	5	1	5	1	5	1	92.5
9		3	1	4	2	4	1	5	3	4	1	80
10		4	1	5	1	4	2	4	2	5	2	85
11		4	4	4	2	4	3	4	3	4	2	65
Average		3.90	1.82	4.18	1.81	4.09	1.73	4.36	2.18	4.27	1.73	78.64

The final score for the SUS questionnaire was 78.64. Any score above 68 is considered above average [49].

Another parameter that can be evaluated, is learnability, which correlates to questions 4 and 10 and can be calculated by adding the average of the scores of both questions. Since these generate a negative response, their scores must be subtracted to five. Then the result is multiplied by 12.5, which will result in a number between 0 and 100.

$$Learnability = 12.5 \cdot ((5 - x_4) + (5 - x_{10})) \tag{4.2}$$

where x_4 is the score of the fourth question and x_{10} is the score of the tenth question. The learnability result was 80.75, which is a positive result.

The participants of this questionnaire were divided by age and gender as follows (see Fig. 4.1 and Fig. 4.2):

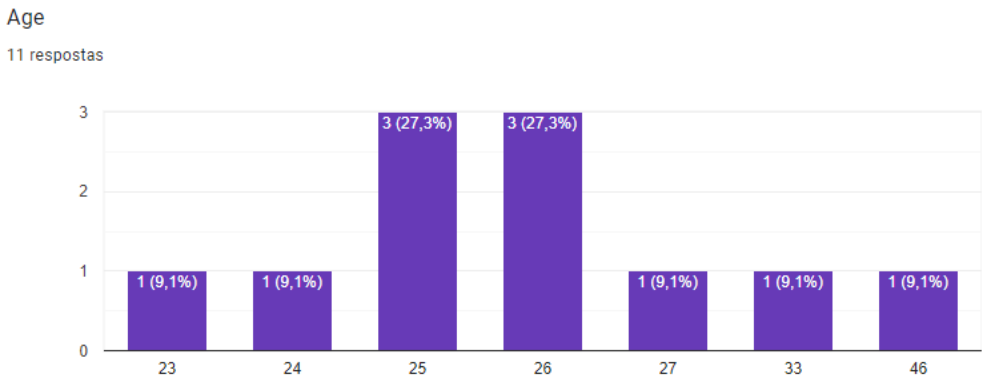


Figure 4.1: Participant age statistics.

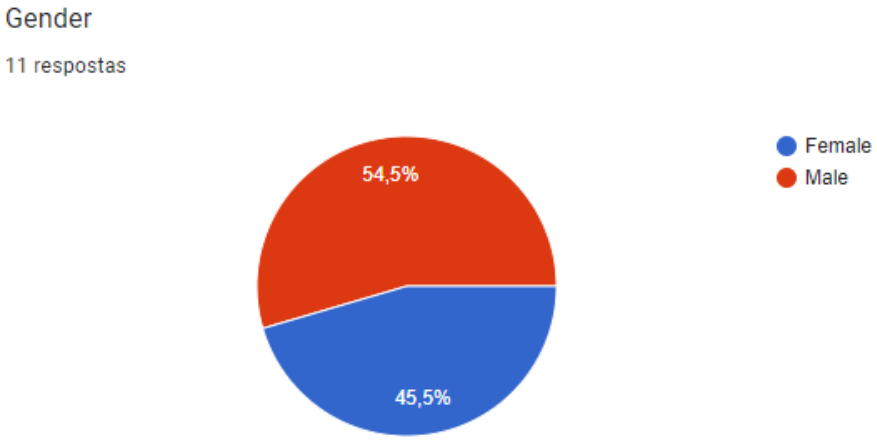


Figure 4.2: Participants gender statistics.

4.2 Correctness of the game

In order to evaluate the performance of the serious game, important values were registered and analysed graphically. The values chosen for this purpose were: the force exerted by the player on the pedals in newtons, the velocity of the rigid body in m/s and the altitude of the bicycle in meters, which is given by the "y" coordinate values, since in Unity's coordinate referential, this coordinate represents the "up" direction (see Fig.4.3).

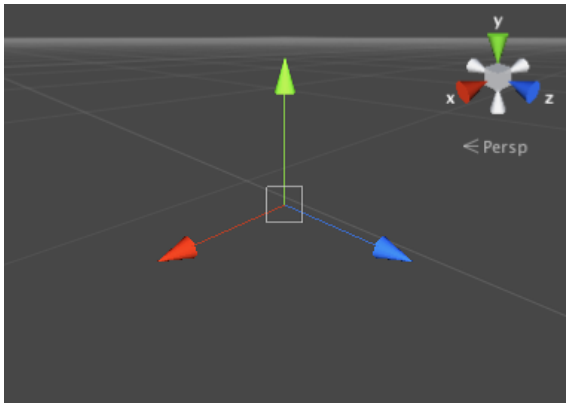


Figure 4.3: Unity's coordinate system.

However, unlike the other values, the velocity of the rigid body couldn't be extracted directly, since this value is in the form of a 3D vector which contains the velocity along each one of the axis coordinates. To obtain the magnitude of the velocity vector, the following equation was applied:

$$v = \sqrt{v_x^2 + v_y^2 + v_z^2} \tag{4.3}$$

where v is the magnitude of the velocity vector, v_x is the velocity in the x coordinate, v_y is the velocity in the y coordinate and v_z is the velocity in the z coordinate.

These values were registered during periods of 100 seconds, for three different levels of resistance, levels one, five and ten and with the mass of the rigid body at 79 kg, and in the same trajectory. We will analyse the effects of the resistance levels on the bicycle's performance, and observe if the bicycle behaves correctly on different levels of force and altitude. For the resistance: level of one, the graphic obtained is as follows (see Fig.4.4).

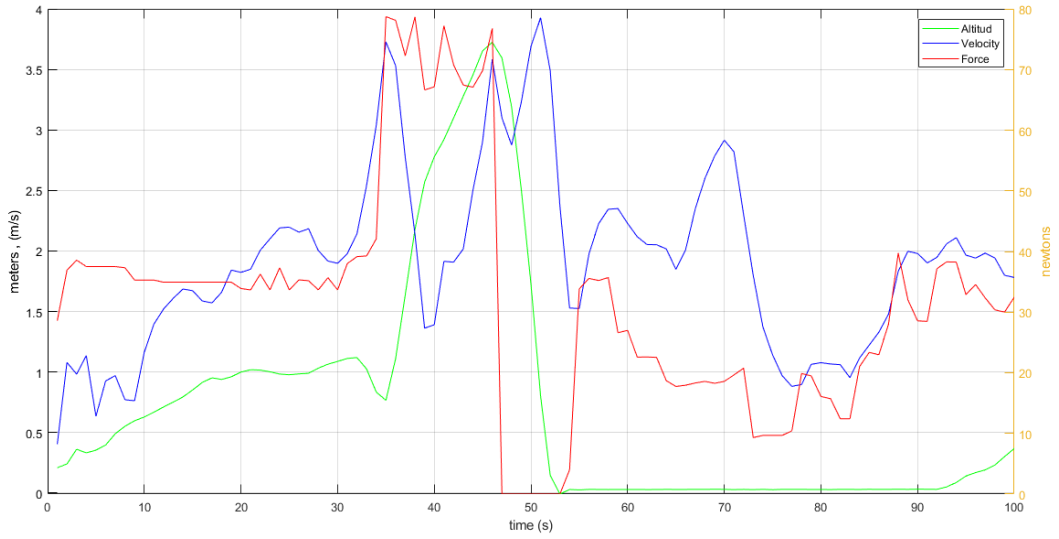


Figure 4.4: Values of force, velocity and altitude at level one for a 100 seconds ride.

In this graphic, it is possible to observe, at a horizontal path, the velocity's magnitude and the force exerted on the pedals are proportional to one another, as can be seen at the interval of 80 to 90 seconds. As the amount of force rises so does the velocity. On the other hand, the velocity's magnitude and the altitude gradient behave in the opposite way which can be observed in the time interval from 35 to 45 seconds, where the altitude rises and for an almost constant amount of force, the velocity decreases.

It can also be seen that when the altitude presents a negative slope the velocity's vector magnitude increases even if the force exerted on the pedals is zero, which can be seen from seconds 45 to 53.

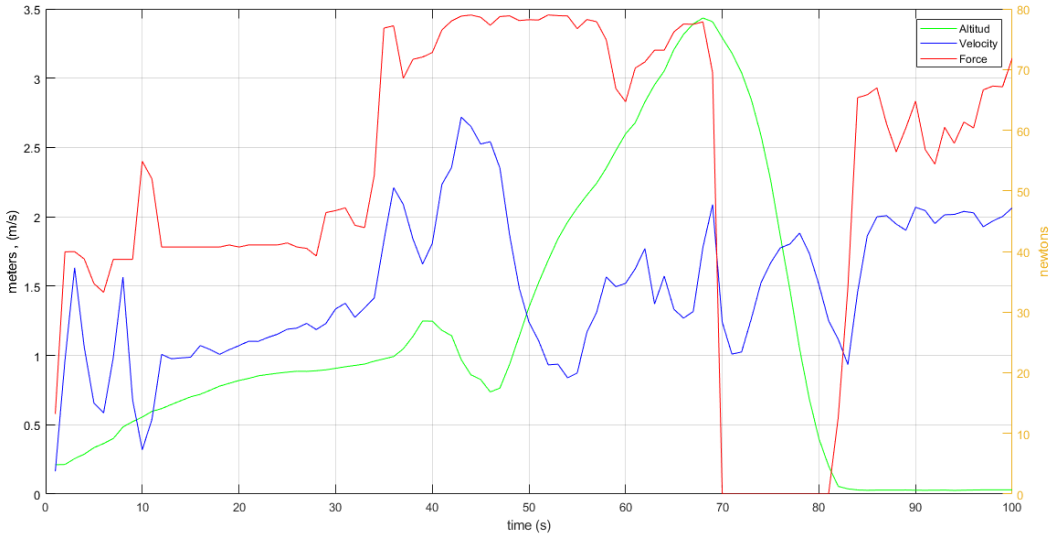


Figure 4.5: Values of force, velocity and altitude with resistance at level five for a 100 seconds ride.

With the resistance at level five, (see Fig.4.5), the bicycle behaves similarly to the lower resistance, although some differences can be observed. For example, the velocity's vector magnitude is lower compared to the former example, reaching a maximum of 2.7 m/s compared to the previous value of 3.9 m/s for similar values of the pedal's force.

Additionally, for the same time period, the highest peak of the altitude is at approximately 70 seconds whereas in the previous graphic, it was at approximately 50 seconds, meaning the length of the path travelled by the bicycle was shorter.

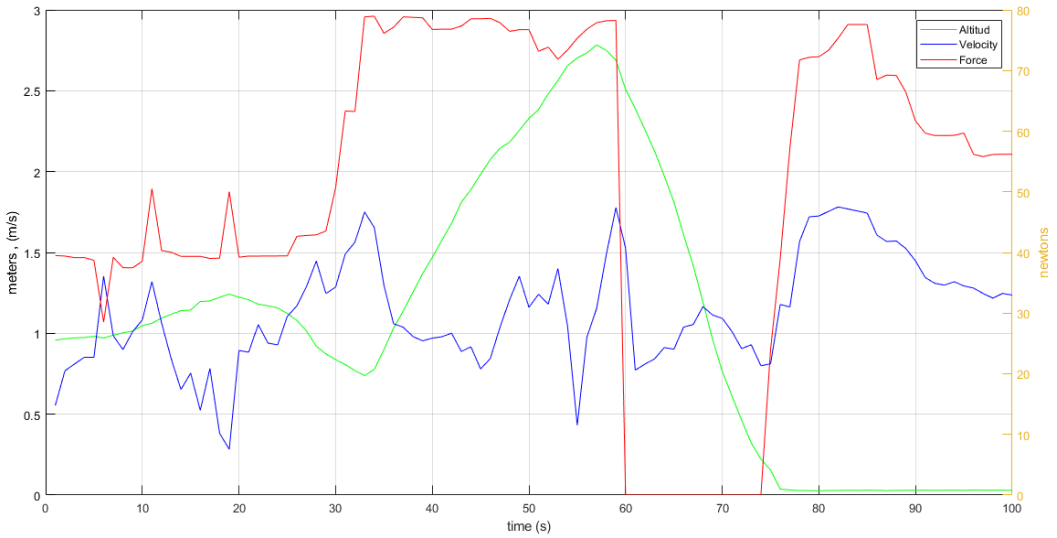


Figure 4.6: Values of force, velocity and altitude with resistance at level ten for a 100 seconds ride.

Finally, with the resistance level at ten (see Fig.4.6), the maximum value for the velocity's vector magnitude was $1.8m/s$, showing a decrease in speed for similar force values, as intended. Moreover, the maximum velocity achieved at the negative slope of the altitude line is much lower compared to the previous versions.

To conclude, in all of these examples, it is possible to observe a coherent behaviour between the variables displayed on the three graphics, as well as the evident effect of the resistance levels on the velocity of the bicycle and therefore the time needed to travel the route.

5 Conclusions

This dissertation presents a method of motivating and assisting patients during the course of physical rehabilitation therapy. After researching the state of the art in the fields of physiotherapy and serious games, Unity was chosen as the main developing tool for this work.

In the presented serious game, the users can choose between three distinct working modes, and personalize their workout experience, by adapting the game functionalities to their unique case.

During experimental validation, in order to evaluate the performance and usability of this work, a SUS questionnaire was carried out which achieve positive scores in both usability and learnability metrics. Also, a graphical evaluation was performed which showed the correct response of the bicycle model in relation to different inputs and slope of terrain. For lower resistance or in the presence of a negative slope relative to the altitude line, the magnitude of the velocity vector increased, whereas, in the presence of a positive altitude slope or higher resistance level selected the velocity decreased. It is also possible to obverse that the higher the amount of force exerted the higher the velocity.

5.1 Future Work

This work can still be improved in different ways. One such example would be the creation of bigger and different gaming scenarios, for example, city environments, beach environments and others. Since these advances require a higher processing capacity, this functionality was not possible to be implemented due to hardware limitations.

In addition, access to a database that stores patient information and possesses a network connection which would allow fluent communication between the user and the physiotherapist could be a possible improvement as well.

Finally, the most crucial improvement would be the connection of this serious game to the

motor-assisted static bicycle it was designed for, this would allow for better testing options and, consequently, fine-tuned upgrades.

Bibliography

[1] M. Forhan and S. V. Gill, "Obesity, functional mobility and quality of life", *Best Practice & Research Clinical Endocrinology & Metabolism*, vol. 27, no. 2, pp. 129–137, 2013.

[2] M. Rantakokko, M. Mänty, and T. Rantanen, "Mobility decline in old age", *Exercise and sport sciences reviews*, vol. 41, no. 1, pp. 19–25, 2013.

[3] M. Mollaoğlu, F. Ö. Tuncay, and T. K. Fertelli, "Mobility disability and life satisfaction in elderly people", *Archives of gerontology and geriatrics*, vol. 51, no. 3, e115–e119, 2010.

[4] I. G. Van De Port, G. Kwakkel, I. Van Wijk, and E. Lindeman, "Susceptibility to deterioration of mobility long-term after stroke: A prospective cohort study", *Stroke*, vol. 37, no. 1, pp. 167–171, 2006.

[5] S. Calthorpe, E. A. Barber, A. E. Holland, *et al.*, "An intensive physiotherapy program improves mobility for trauma patients", *Journal of Trauma and Acute Care Surgery*, vol. 76, no. 1, pp. 101–106, 2014.

[6] D. Zidarov, B. Swaine, and C. Gauthier-Gagnon, "Quality of life of persons with lower-limb amputation during rehabilitation and at 3-month follow-up", *Archives of physical medicine and rehabilitation*, vol. 90, no. 4, pp. 634–645, 2009.

[7] E. Jones, J. Pike, T. Marshall, and X. Ye, "Quantifying the relationship between increased disability and health care resource utilization, quality of life, work productivity, health care costs in patients with multiple sclerosis in the us", *BMC health services research*, vol. 16, no. 1, pp. 1–9, 2016.

[8] A. Bracke, G. Domanska, K. Bracke, *et al.*, "Obesity impairs mobility and adult hippocampal neurogenesis", *Journal of experimental neuroscience*, vol. 13, p. 1 179 069 519 883 580, 2019.

[9] H. K. Vincent, K. R. Vincent, and K. M. Lamb, "Obesity and mobility disability in the older adult", *Obesity reviews*, vol. 11, no. 8, pp. 568–579, 2010.

[10] M. Matinolli, J. Korpelainen, R. Korpelainen, K. Sotaniemi, V.-M. Matinolli, and V. Myllylä, "Mobility and balance in parkinson's disease: A population-based study", *European Journal of Neurology*, vol. 16, no. 1, pp. 105–111, 2009.

[11] G. S. Gilmour, G. Nielsen, T. Teodoro, *et al.*, "Management of functional neurological disorder", *Journal of Neurology*, vol. 267, pp. 2164–2172, 2020.

[12] S. Tyson and L. Connell, "The psychometric properties and clinical utility of measures of walking and mobility in neurological conditions: A systematic review", *Clinical rehabilitation*, vol. 23, no. 11, pp. 1018–1033, 2009.

[13] R. E. Pecoraro, G. E. Reiber, and E. M. Burgess, "Pathways to diabetic limb amputation: Basis for prevention", *Diabetes care*, vol. 13, no. 5, pp. 513–521, 1990.

[14] P. A. Lazzarini, S. R. O'Rourke, A. W. Russell, D. Clark, and S. S. Kuys, "What are the key conditions associated with lower limb amputations in a major australian teaching hospital?", *Journal of foot and ankle research*, vol. 5, pp. 1–9, 2012.

[15] C. J. Brown, R. J. Friedkin, and S. K. Inouye, "Prevalence and outcomes of low mobility in hospitalized older patients", *Journal of the American Geriatrics Society*, vol. 52, no. 8, pp. 1263–1270, 2004.

[16] D. T. Wade, F. M. Collen, G. F. Robb, and C. P. Warlow, "Physiotherapy intervention late after stroke and mobility.", *British Medical Journal*, vol. 304, no. 6827, pp. 609–613, 1992.

[17] J. Viitanen, J. Suni, H. Kautiainen, M. Liimatainen, and H. Takala, "Effect of physiotherapy on spinal mobility in ankylosing spondylitis", *Scandinavian Journal of Rheumatology*, vol. 21, no. 1, pp. 38–41, 1992.

[18] C. Wiles, R. Newcombe, K. Fuller, *et al.*, "Controlled randomised crossover trial of the effects of physiotherapy on mobility in chronic multiple sclerosis", *Journal of Neurology, Neurosurgery & Psychiatry*, vol. 70, no. 2, pp. 174–179, 2001.

[19] D. K. A. Singh, N. A. Mohd Nordin, N. A. A. Aziz, B. K. Lim, and L. C. Soh, "Effects of substituting a portion of standard physiotherapy time with virtual reality games among community-dwelling stroke survivors", *BMC neurology*, vol. 13, no. 1, pp. 1–7, 2013.

[20] R. López-Liria, D. Checa-Mayordomo, F. A. Vega-Ramırez, A. V. Garcıa-Luengo, M. Á. Valverde-Martınez, and P. Rocamora-Pérez, "Effectiveness of video games as physical treatment in patients with cystic fibrosis: Systematic review", *Sensors*, vol. 22, no. 5, p. 1902, 2022.

[21] T. Susi, M. Johannesson, and P. Backlund, "Serious games: An overview", 2007.

[22] T. Martins, V. Carvalho, and F. Soares, "A serious game for rehabilitation of neurological disabilities: Premilinary study", in *2015 IEEE 4th Portuguese Meeting on Bioengineering (ENBENG)*, IEEE, 2015, pp. 1–5.

[23] A. Oikonomou and D. Day, "Using serious games to motivate children with cystic fibrosis to engage with mucus clearance physiotherapy", in *2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems*, IEEE, 2012, pp. 34–39.

[24] R. Baranyi, R. Willinger, N. Lederer, T. Grechenig, and W. Schramm, "Chances for serious games in rehabilitation of stroke patients on the example of utilizing the wii fit balance board", in *2013 IEEE 2nd International Conference on Serious Games and Applications for Health (SeGAH)*, IEEE, 2013, pp. 1–7.

[25] D. Ferreira, R. Oliveira, and O. Postolache, "Physical rehabilitation based on kinect serious games", in *2017 Eleventh international conference on sensing technology (ICST)*, IEEE, 2017, pp. 1–6.

[26] A. Andrade, "Game engines: A survey", *EAI Endorsed Transactions on Serious Games*, vol. 2, no. 6, 2015.

[27] Unity, *Input system, introduction*, `https://docs.unity3d.com/Packages/com.unity.inputsystem@1.5/manual/index.html`.

[28] Unity, *Supported input devices*, `https://docs.unity3d.com/Packages/com.unity.inputsystem@1.5/manual/SupportedDevices.html`.

[29] Unity, *Input system, workflows*, `https://docs.unity3d.com/Packages/com.unity.inputsystem@1.5/manual/Workflows.html`.

[30] Unity, *Input system, actions*, `https://docs.unity3d.com/Packages/com.unity.inputsystem@1.5/manual/Actions.html`.

[31] Unity, *Unity, prefabs*, `https://docs.unity3d.com/Manual/Prefabs.html`.

[32] Unity, *Unity core packages, unity ui*, https://docs.unity3d.com/Manual/com.unity.ugui.html.

[33] Unity, *Important classes*, https://docs.unity3d.com/Manual/ScriptingImportantClasses.html.

[34] Unity, *Scenes*, https://docs.unity3d.com/Manual/CreatingScenes.html.

[35] Unity, *Gameobjects manual*, https://docs.unity3d.com/Manual/GameObjects.html.

[36] Unity, *Rigidbody physics*, https://docs.unity3d.com/Manual/rigidbody-physics-section.html.

[37] Unity, *The transform component*, https://docs.unity3d.com/Manual/class-Transform.html.

[38] Unity, *Rigidbody component reference*, https://docs.unity3d.com/Manual/class-Rigidbody.html.

[39] Unity, *Rigidbody collision*, https://docs.unity3d.com/Manual/rigidbody-configure-colliders.html.

[40] Unity, *Mesh collider component reference*, https://docs.unity3d.com/Manual/class-MeshCollider.html.

[41] NVIDIA, *Physx 3 vehicles sdk*, https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Vehicles.html.

[42] Unity, *Wheel collider component reference*, https://docs.unity3d.com/Manual/class-WheelCollider.html.

[43] P. Minarik, *How much do road bike wheels weigh? data from more than 400 wheelsets*, https://www.cyclistshub.com/road-bike-wheels-statistics/#:~:text=Road.

[44] Unity, *Wheel collider, force app point distance*, https://docs.unity3d.com/ScriptReference/WheelCollider-forceAppPointDistance.html.

[45] Unity, *Important classes, mathf*, https://docs.unity3d.com/Manual/class-Mathf.html.

[46] Unity, *Quaternion, "quaternion.euler"*, https://docs.unity3d.com/ScriptReference/Quaternion.Euler.html.

[47] Unity, *Scenes*, https://docs.unity3d.com/Manual/CreatingScenes.html.

[48] Unity, *Important classes, "scriptableobject"*, https://docs.unity3d.com/Manual/class-ScriptableObject.html.

[49] A. Assila, H. Ezzedine, *et al.*, "Standardized usability questionnaires: Features and quality focus", *Electronic Journal of Computer Science and Information Technology*, vol. 6, no. 1, 2016.

[50] K. Betteridge, *What every uxc client should know about sus scores*, https://www.bentley.edu/centers/user-experience-center/what-every-client-should-know-about-sus-scoresl.