ARTHUR CARVALHO DE QUEIROZ
ARTHUR PEDRI TREVISOL

A PEDESTRIAN STREET CROSSING VIRTUAL REALITY SIMULATOR FOR MOBILE DEVICES DEVELOPED IN UNITY

(*pre-defense version, compiled at December 6, 2021*)

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Eduardo Todt.

CURITIBA PR

2021

RESUMO

Cada vez mais o número de carros na rua está crescendo, e por isso os sistemas de prevenção de acidentes e proteção do motorista estão cada vez mais avançados, mas a quantidade de acidentes sofridos por pedestres ainda é muito alarmante. Vários centros de treinamento e autoescolas já precisam ter ferramentas de treinamento antes de levar o motorista para as ruas, como simuladores de trânsito, mas o outro lado dessa equação é muito negligenciado, o lado do pedestre. Com a motivação de treinar pedestres em uma simulação real de trânsito e coletar informações sobre o processo de tomada de decisão do pedestre, aliada ao recente avanço na tecnologia de realidade virtual, em termos de qualidade e acessibilidade, nos propomos a criar uma aplicação de realidade virtual para Android, sendo usada com óculos de realidade virtual, para que possa ser feita uma avaliação e treinamento dos maiores problemas e erros feitos por pedestres em uma situação realista de trânsito. O desenvolvimento foi separado em três partes: a *contrução do mundo*, com a definição de todos os objetos estáticos com que futuramente o usuário irá interagir; *Interações e funcionalidades*, o desenvolvimento do software para permitir todas as capacidades esperadas; *Entrada e saída*, possibilitando a interação com a simulação através de um controle, e extraindo os dados da simulação em um formato estruturado para posterior análise. Como resultado, a aplicação permite que qualquer pessoa com um celular com a versão 4.4 do Android e um *headset* de realidade virtual possa experienciar a simulação e coletar os dados a respeito das decisções de cada usuário de forma segura, replicável e acessível.

Palavras-chave: Realidade Virtual, Simulação de Trânsito, Pedestre, Unity, Android, Aplicação, VR, HMD, App

ABSTRACT

The number of cars on the streets is growing every day, and thus accident prevention and protection systems are getting more advanced. However, the number of accidents suffered by pedestrians is still very alarming. Various training centers and driving schools already require training tools before taking drivers to the road, like traffic simulators. Still, the other side of this equation is often neglected, the side of the pedestrian. With the motivation to train pedestrians in a realistic traffic situation, we set out to develop a virtual reality application on Android to be easily accessible with store-bought virtual reality goggles. The simulator will allow evaluation and training of the most significant problems and mistakes made by pedestrians in a realistic traffic situation. The development was separated into three main sections: the *world building*, defining all the static objects that would later be interacting with the user; *Interactions and functionality*, fleshing out the back-end of the application to allow full functionality; *Input and output*, enabling controller input for interactivity and outputting simulation data to a workable format for further analysis. As a result, the application allows for anyone with a mobile device running Android 4.4 and a basic VR headset to experience the simulation and collect data on the decisions made by every participant in a safe, replicable, and accessible way.

Keywords: Virtual Reality, Traffic Simulation, Pedestrian, Unity, Android, Application, VR, HMD, App

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

DINF	Departamento de Informática
PPGINF	Programa de Pós-Graduação em Informática
UFPR	Universidade Federal do Paraná
VR	Virtual Reality
AR	Augmented Reality
AV	Augmented Virtuality
XR	Extended Reality
HMD	Head-mounted display
MPE	Multi-projected environment
IMU	Inertial measurement unit
3-DoF	Three degrees of movement
6-DoF	Six degrees of movement
SDK	Software development kit
APK	Android application package
3D	Three dimensions
2D	Two dimensions
LTS	Long-term support
LOD	Level of Detail
Km/h	Kilometers per hour
JSON	JavaScript Object Notation
C#	C Sharp
CSV	Comma-separated values
UI	User interface
S	Seconds

CONTENTS

1 INTRODUCTION

The first time most children learn about how to interact with traffic is early in school, and while learning about road signs and the rules of the road is very important, so is learning how to apply this knowledge to the real world. While drivers get further education on traffic laws, correct behavior on the road, and actual experience when they apply for a license, pedestrians are usually left with only the education they received in their youth and are forced to learn how to deal with real situations when the need arises. This lack of further education makes it so incorrect crossing behavior is a major factor in most cases of pedestrians being run over (Torquato Steinbakk and Bianchi, 2010). Having realistic experiences for learning pedestrians, as we do for learning drivers, is as unviable as it is dangerous; therefore, most people grow up without learning to gauge many of the uncertainties of traffic and are ill-equipped to deal with them because of this lack of experience.

Research done with teenagers reveals unsatisfactory results, displaying their inability to make proper decisions and the risk that they put themselves in when confronted with traffic (Weiss, 2019). Brazilian drivers, in particular, are exceptionally disrespectful towards pedestrians and crosswalks, in general, even though the fines for driving while someone is crossing the street are steep, and doing so will suspend the drivers' license (R7, 2021).

One of the methods of teaching correct habits and behavior is to make use of simulation rigs. For decades, this has been a staple of the aviation world, as getting in complicated situations is extremely unforgiving on inexperienced airplane pilots. As driving does not require such in-depth knowledge and extensive training, simulators have not been historically used for teaching driving. However, as more schools begin to adhere to this practice, the benefits of this method can be better displayed. In Brazil, by law, learners have the option of using a simulator in place of a few real driving hours when getting their license (MI/CNT, 2019), as seen in figure 1.1.

Simulations have many benefits over conducting experiments in the real world. The reason why they have been getting more widespread use could be that having a specific and adaptable application that can be used in any pre-programmed scenario is highly advantageous. One of the significant issues with comparing results in complex and unpredictable situations is that other factors often lead to the observed results. This is much less of an issue with computer simulations because the world can be literally "done over," eliminating most variables, which is impossible in reality. Financially, in a general sense, investing in these solutions is usually a front-loaded endeavor, as most of the cost is concentrated on the initial development of the technology. The rest is used to maintain it and then further train subjects on the simulations, which has a much lower cost.

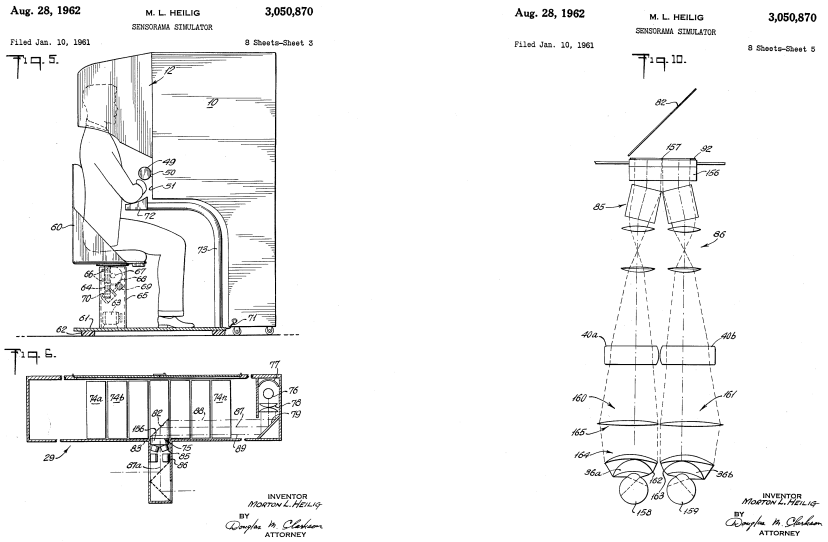
Virtual Reality is an interface for simulations that grants users an immersive and interactive experience through graphics and other sensory stimuli. This immersion is primarily

Figure 1.1: Simulator rig used by learning drivers. Reference: (G1, 2013)

achieved by using goggles to direct each eye to a different screen, creating the stereoscopic view. This solution helps our brain perceive two-dimensional screens as one three-dimensional image and tricks our brain into thinking we are in the real world, not in a computer simulation.

The origin of VR is not certain, but as early as the 1960s, Heilig (1962) created a patent of what would be a cabin that simulates an individual's sensory experience by providing separate parallel images of the same object from different perspectives, the same effect that gives us the capacity to observe the third dimension. The idea of replicating real sensations has only evolved since then. As technology advanced and screens got lighter, the first examples of Head-mounted displays were seen, where the subject could wear a device that resembles a partial helmet with a simple screen on their head. When modern cellphones acquired the inertial measurement units required to transform HMDs into proper VR headsets, developing mobile applications with VR capabilities became a reality.

As technologies that allow us to experience VR are becoming more accessible, and observing the level of immersion that can be achieved on a simple handheld device and cheap VR goggles, we developed an Android application to aid in training and to collect information with minimal infrastructure requirements. In the following pages of this work, we will be talking more in-depth about VR, the current technologies and studies that have been done, our decisions regarding the simulation, and the implementation of our pedestrian simulator for urban street crossing.



(a) Simulation cabin.

(b) Separate images of the same object.

Figure 1.2: Sensorama simulator patent. Reference: (Heilig, 1962)

1.1 MOTIVATION

The motivation for creating this work came from the lack of a simpler tool to help in studies of pedestrian behavior in traffic, primarily aimed toward younger audiences that do not have the ample experience of dealing with traffic that adults have. Collecting data on pedestrian habits is not trivial. Besides the inherent risk of dealing with moving vehicles, the environment itself can't be easily replicated. These limitations make simulations an effective tool to aid in said data collection. The primary issue with these kinds of simulations would be the infrastructure and development costs associated with VR. A market-grade VR headset is a sizable investment, and because of that, the data collection can't be easily scaled. While not the best in terms of quality or immersion, we found that applications that can run on mobile devices with basic VR headsets, like the one seen in figure 1.3, are a great alternative to scalability, accessibility, and price compared to cutting-edge VR headsets. This work may become a valuable tool to researchers looking to study traffic education or human behavior when interacting with vehicles and urban scenarios in a safe, replicable, and accessible way.

1.2 OBJECTIVES

This project seeks the development of an Android application that will use a controller to move the player and the mobile device itself as the screen to simulate an urban environment in which the user must determine the best place and time to cross the street. The application will also save a collection of indicators and information about each execution for further study. As a result, this

Figure 1.3: One of the most basic VR headsets, the Google Cardboard. Reference: (Google, 2021b)

application should supply a tool for aiding and incentivizing research and education on traffic safety.

1.3 METHODOLOGY

The simulation was developed using the following methodology:

World Building: Creating the main space of the simulation, importing pre-made models of complex objects, adding VR functionality to the player camera, defining all the static objects that would later be interacting with the user.

Interactions and Functionality: Adding functionality to the static world by allowing player control and decision making, car movement, player and world configuration, traffic control, interactions of player and cars, menu navigation, and street crossing capabilities.

Input and Output: Controller input configuration, collecting information of the current session and outputting collected simulation data to a workable format for further analysis.

With the application compiled for Android and having a simple VR headset, it's possible to experience the simulation and collect data on the decisions made by the user, which can be later extracted and analyzed for common decisions and patterns of behavior.

1.4 STRUCTURE

This work is structured in the following way:

1. Introduction: This chapter.

2. **Theoretical Foundation:** What is VR and how is it different from other interfaces, the cutting edge and VR in the mainstream, how our method differs from the cutting edge. Main technologies used for the development of the simulator: The Unity game engine functionality and concepts that will be used in the following chapters, the Google VR Software Development Kit and how it relates to Unity, the Android platform, and why we chose it. Arguments for choosing these technologies over their competitors and their advantages.

3. **Related Works:** Introduction of previous works that relate and use Traffic simulators and Virtual Reality.

4. **Proposal:** The software requirements used as a guideline for the development, the decisions, and subsequent implementation of the solutions to the requirements of the simulation in terms of the simulated world, the movement of the player, the traffic control, the interaction of player and environment, the UI and data collection.

5. **Experiments and Results:** Experimentation and subsequent analysis of the results for validation and conclusions of the simulation and the involved systems.

6. **Discussion and Conclusion:** Discussion about the possible contributions of this software in certain fields, conclusions about the methodology and effectiveness of this software solution, and possible future works and modifications.

2 THEORETICAL FOUNDATIONS

This chapter highlights the main components and platforms used to make the application possible and elaborates on their functionality, use, and decision-making on the choice of technology and further explanations.

2.1 EXTENDED REALITY

The Extended Reality or XR is an umbrella term that includes everything from actual reality to complete virtual reality. It can be used to refer to any form of computer-human interaction that extends the fundamental experiences and senses that humans have (Goode, 2019).

2.1.1 The Reality-Virtuality Continuum

In their article, Milgram et al. (1994) suggests a continuum (figure 2.1) that spans from the real environment to the virtual environment, with everything in between being part virtual and part real. There are places in this supposed continuum that have been defined and expanded upon to better communicate the purpose and capabilities of a piece of software. Here are some of the more commonly recognizable terms that we find in this continuum:

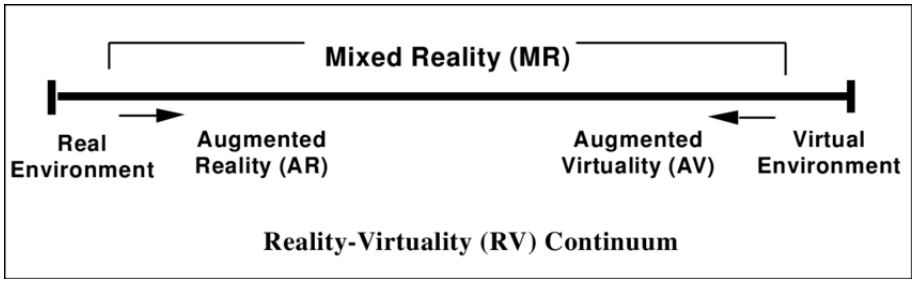


Figure 2.1: The Reality-Virtuality Continuum. Reference: (Milgram et al., 1994)

AR Augmented Reality is the term used to refer to when virtual objects and functionality are enhancing reality. Usually, this is actualized by using a combination of a screen and a camera, where real-time footage captured by the camera is most of what is being displayed. AR is achieved when this footage is enhanced with virtual objects, image filters, additional information, or any other data that is not captured or otherwise present in the real world. One of the initial uses of AR was in the pilot's heads-up displays, where the airplane pilot would benefit from having information displayed on the glass they were looking through instead of the instruments below. Nowadays, AR has seen use in various entertainment applications, such as camera filters and geo-location-based games (Google, 2021a).

AV Augmented Virtuality is in many ways the inverse of AR. The idea is to enhance virtual reality with real-world objects by scanning and merging these objects with the virtual world. This term is much less used because it is very similar to regular virtual reality. As the use of models and textures taken from the real world becomes more prominent, the line between AV and VR is increasingly blurred.

2.2 VIRTUAL REALITY

The definition of Virtual Reality comes parallel to the definition of actual reality, that is, an artificial, computer-generated, 3D environment that can be experienced with our senses, such as sight and hearing, and where one's actions determine a portion of what happens in that environment (ISAR, 2018).

2.2.1 Characteristics

While not having a rigorous definition, VR is generally thought of as a simulated experience with specific characteristics that elevate it to the idea of a different (virtual) "reality."

Computer-generated A virtual reality must be generated with a computer and in 3D because the more closely it resembles the real world, the more the user will be able to immerse themselves in it.

Observable The user must have the ability to observe the world from their own perspective without being locked to a specific angle or position. That is to say when in a particular place in space, the simulation must allow for everything around it to be observable by changing directions in the same way that this is possible in reality.

Interactive As it happens in reality, people should have the capacity to influence and alter the world according to their decisions and interactions. This capacity is what distinguishes VR from an immersive movie, for example.

Immersive The ability to convince the user that they are in a different reality is one of the main goals of the VR experience. While this will not always be perfectly achievable because of various factors like input lag, lower visual quality, and lack of smell, as the simulation gets closer to providing an experience similar to reality, the user gradually comes to accept the simulation as real in their minds. This phenomenon can be observed when the simulation throws an object directly into the user's face, causing an involuntary reaction to move their head out of the way.

Feedback Because VR is *observable* and *interactive*, the user must receive feedback from what they have interacted with or observed in the world. This feedback is usually presented visually by altering the display and in auditory form by sending the appropriate sounds

to the user. But these are not all the possible responses. Depending on the hardware, the user can also receive haptic [1] feedback when interacting with an object; this can be achieved by using gloves with multiple sensors or simply by using a controller with a vibration function.

2.2.2 Common Uses

As VR can simulate the real world, we can find a broad range of uses for such a technology (Bardi, 2021).

2.2.2.1 Visualization Tool

Visualization tools for science, engineering, architecture, and commerce are widely used. They can aid in understanding the subject at hand, such as mechanical components, floor tile placement, and physically unavailable vehicles for purchase. VR adds to these tools by allowing the visualization of depth and having a more intuitive way of navigating the different perspectives of what is being visualized instead of using devices such as a mouse and keyboard to translate their movements into natural head movements.

2.2.2.2 Education and Training

In basic education scenarios, the level of immersion provided by using VR aids in concentration and content absorption, but the real advantages come in the form of inaccessible or otherwise difficult to replicate education and training. Some of the beneficiaries of this aspect are aviation, medicine, and the military. The price of failure in these particular fields is significantly elevated. Consequently, they tend to appreciate the opportunity to improve their skills in a realistic, immersive, but ultimately low-stakes environment that simulating in VR can provide (Balasubramanian, 2021).

2.2.2.3 Treatment

Beyond helping and training doctors, VR is also being used by patients to help with post-traumatic stress disorder and mental illness. Typical uses include simulating stressful or triggering situations for the patients in controlled conditions and overcoming phobias without being exposed to real-world complications. With a high-quality setup, the immersion achieved by these simulations can convince the brain that the situation is real and greatly assist in treatment.

2.2.2.4 Gaming

As the interest in VR grows, naturally, the demand for VR games will follow. This demand is a large part of the driving force in new market grade VR hardware, software development

[1] Haptic technology or 3D touch refers to creating the experience of touch by applying vibration, forces, or motions to the user.

solutions, and a more significant focus on VR technologies. Games include a wide range of possible software, and they encourage the creative freedom that allows for a computer to properly generate and simulate virtual reality.

2.2.3 Our use of VR

We decided on the use of VR for our simulation primarily because of the necessity of immersion. It's easier to treat a street crossing simulator as a game and thus fail to collect relevant data if the participants do not feel somewhat like they are in a real-world situation. While we sacrificed some immersion in favor of scalability, affordability, and other factors that will be further outlined below, we feel that the level of immersion our simulation can provide is sufficient to achieve similar decision-making in our participants as they would have in reality.

2.3 HARDWARE FOR VR

To achieve immersion and interact with virtual realities, it is of utmost importance to have the proper hardware that allows for visual and auditory representation of the virtual environment. A variety of capable hardware has been created for this purpose, and they can be generally classified into two groups: Head-mounted displays and multi-projected environments.

2.3.1 Multi-projected environments

MPEs achieve the immersion effect necessary for VR by using projectors and screens in a preferably darkened room to make up for the visual component of the simulation. While impressive in its own right, this method lacks the depth perception possible when using separate displays for each eye. Besides the visual component, MPEs can include sensors around and on the user, accounting for movement tracking and possibly even a treadmill for simulating walking, as was done by Kazutaka Mitobe and Yoshimura (2012). Unfortunately, even though it is more expensive to build, this method seems to lag behind head-mounted displays in both immersion and interaction possibilities.

2.3.2 Head-mounted displays

The simplest possible HMD is, as the name suggests, a simple screen that can be mounted on the head. More sophisticated HMDs will include tracking changes in angles and orientation of the head and later position tracking for reading the position of the wearer's head (Sutherland, 1968). Further improvements will see the ability to measure and record the wearer's gaze and track hand movement. For our purposes, we will be using HMDs that can track angles and orientation of the head but can't detect head position, gaze, and hand movement. The main ways in which head movement is tracked and that a rigid object can be moved in three-dimensional space can be described in degrees of freedom, as illustrated in figure 2.2.

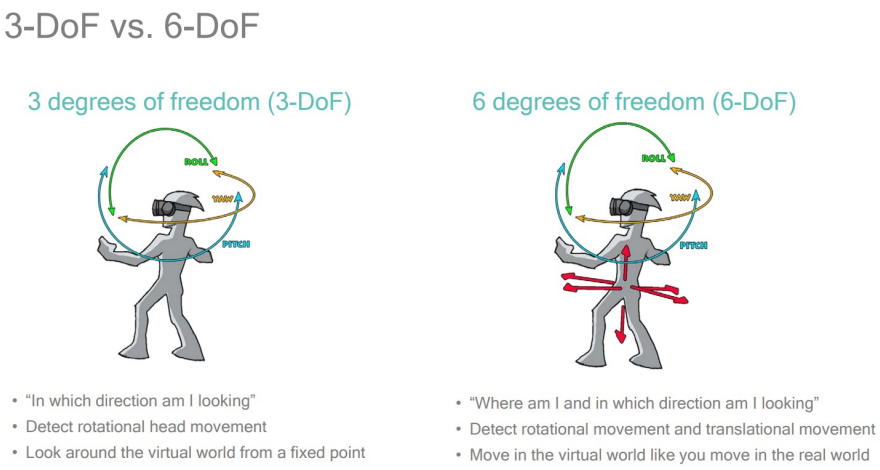


Figure 2.2: 3-DoF vs. 6-DoF. Reference: (Qualcomm, 2017)

3-DoF Three degrees of freedom allow tracking only the device's rotation on the three axes, and the movements are named pitch, yaw, and roll. This means that if the user were to lower their head or walk around, the sensors would not capture any change, and thus the simulation would not be updated in any way. Rotation is tracked mostly through inbuilt sensors, such as the accelerometer that measures instant acceleration, the gyroscope that measures orientation and angular velocity, and the magnetometer that measures magnetic fields and their interactions.

6-DoF Six degrees of freedom allow for translational movement in addition to rotation along those axes, which means that, in the idea of an HMD, it can track the position of the head in space when it moves. Translational movement is detected in different ways, depending on the device maker. One of the most common strategies is to use separate position tracking components placed further away on opposite sides of the device that emit a signal to be detected by the HMDs' sensors, usually some form of infrared lasers, so the software can effectively map the space the user is in and their relative movements.

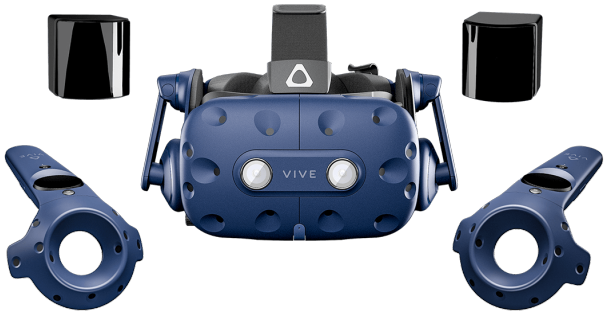


Figure 2.3: The HTC Vive Pro with controllers and position tracking components on the back. Reference: (Vive, 2021)

The more commonly seen HMDs are built with two small high-definition screens that will stereoscopically [2] generate an image for each eye, which the user will be able to see through lenses placed near their eyes, much like glasses (Klein, 2021). The lenses and screens can be combined and supported through either an elastic band or a helmet-like structure (Figure 2.3). A stereo headphone can be an invaluable addition to the setup, and most of the HMDs on the market will feature some integration for better sound. Additional interactivity can be achieved by adding controllers, so the user will have a greater capacity to control their actions and movements. Higher-end HMDs can also feature interactions between the controllers and the positional trackers, effectively adding functional hands with rotation and position to the simulation.

As the market grows, new high-end HMDs are seeing the light of day and becoming patented products of big companies with different specialties and strategies to achieve proper virtual reality. In figure 2.4 we can observe these higher quality and price HMDs.

(a) Meta's Oculus Quest. Reference: (Oculus, 2021)

(b) Sony's PlayStation VR. Reference: (Sony, 2021)

Figure 2.4: Along with the HTC Vive, three of the biggest players in the HMD market.

[2] Stereoscopy or stereo imaging is a technique used for creating the illusion of depth in an image by presenting two images offset by the distance between two eyes, one in the left and one in the right.

2.3.3 Mobile Devices as HMDs

As more mobile devices are made with the sensors necessary for capturing 3-DoF movement (Section 2.3.2), with processing power constantly improving, and as their screens increase in resolution further, we can now use them as the display and most of the hardware for a functioning HMD (Statt, 2014). As this method is now possible, the market has seen the necessity of producing simplistic head mounts that can support most mobile devices, such as the one displayed in figure 1.3. These head mounts are much more affordable than full HMDs, and since most people have access to mobile devices, this seemed to us a better strategy for our purposes.

While not nearly as capable as full HMDs, mobile devices can better fulfill our needs in certain ways:

3-DoF: While not an advantage in terms of immersion, having three degrees of freedom instead of six allows us not to need a recentering function [3]. This is particularly noticeable when starting the simulation with the device away from the head, because the act of bringing it to the head will be a movement recorded by the position trackers with 6-DoF, and will result in the player having a different height or being in a different position than their real selves.

Affordability: If not accounting for the price of a capable mobile device, because most people already own one, the price for the head mounts to be used with them is a fraction of the price of market HMDs. Since they have to be imported to Brazil, taxes will further increase their cost. This is not even considering the cost of the powerful computer needed to run software on the full HMD. They usually will only work as a monitor that requires a video connection with a computer.

Accessibility: Besides the price, another issue with regular HMDs is that they are just not that available for purchase in Brazil and can usually only be found online or in big technology shops. The head mounts we will be using for our project are not restricted in terms of maker or model, so they can be produced locally and are much easier to find and purchase.

Portability: Portability is another factor that we considered, as our hardware allows for a minimal footprint and can be easily transported without much additional concern. The ability to run through the entire process with battery power is also an excellent boon for portability.

Scalabilty: Because the required hardware is not very restricted, it's easy to scale studies in terms of participants. Consequently, more data can be gathered, and the results will be more statistically relevant.

[3] A recentering function will revert any movement or rotation made by the user to the default position.

Ease-of-use: Having the application on mobile will make it easier to access, download, and use. The process of installing software on a mobile device is, in general, more familiar to people.

Comfort: Because our head mount is generic, and the user can choose to run the simulation in a product that better fits their needs, they can adjust for weight, shape, and size. Furthermore, the lack of cables that need to be connected to a computer makes for a much more comfortable experience.

Software updates: Developing for mobile makes updates easily deployable and shortens the time between new versions and their adoption by the user base.

Software bloat: For our project, the only extra piece of software that will be necessary is the Cardboard app. Compared to the various software and driver installations required to use a full HMD, we can see that software bloat is a problem that we also circumvent by choosing to develop for mobile.

2.3.4 Other Hardware

Other hardware that can be used with VR includes mainly haptic gloves, which can greatly enhance the user's sense of touch, and omnidirectional treadmills (Lemon, 2013), which allow the user to walk in 360 degrees without moving. While the gloves are not as interesting for our street crossing simulator, the treadmills certainly are. So much so that on some of the articles we highlighted previously the authors used treadmills to great effect, both regular and omnidirectional. The reason for opting out of using them is mostly the same as for choosing HMDs with mobile devices as screens. They make the simulation much less affordable, portable, accessible, scalable, and even harder to use. While they certainly have advantages, we felt that they were not sufficient to overcome the downsides, and thus the only additional hardware we use is a simple game controller with a joystick and buttons. This solution is much less immersive, but it goes in line with our priorities and continues to adhere to our motivation.

2.4 DEVELOPMENT IN VR

VR Development can be broken down into a few key steps (Wirtz, 2021):

3D Geometry: Because VR is strictly a 3D experience, it is paramount that we can work with three coordinate geometry. This includes positioning, rotation, and scaling of objects with volume.

Rendering: Selectively drawing a 2D image of the 3D geometry, based on the angle and direction of some reference point, usually the eyes of the player's representation in the virtual world. Drawing two slightly displaced images will create the effect of depth in our

brains, one of the most important avenues to create immersion. Light calculations must also be dealt with to render both these images correctly.

3D Modeling: Having the ability to calculate and render in 3D leads to creating higher quality objects. These objects are made to be as faithful to their real-world counterparts as possible, adding another layer to the immersion of the virtual world. This modeling is usually done with very specialized software and has many professionals solely dedicated to it, as it is essentially creating geometric art.

Physics Computation: Adding rules as to how objects interact and render is crucial in creating a cohesive environment for development. These rules will include collision, acceleration, gravity, friction, and other rules that may be relevant to a particular simulation.

Input Handling: Receiving and processing input by all devices the user can access, including sensors and controllers, and translating them into the virtual world in the appropriate ways.

2.5 GAME ENGINES

A Game Engine is a piece of software that implements most, if not all, the functionalities necessary for developing games, with libraries and abstractions to aid in the creation process. All the steps described in section 2.4 to develop for VR are already natively included in an engine capable of creating games in 3D and VR. An engine will provide a friendlier user interface, sound support, and various other features by default (Florian, 2018). The addition of the user interface and built-in functionalities allow for users with less programming knowledge to be capable of developing games without having to acquire in-depth knowledge of programming, math, and some of the other steps necessary for VR development. Besides the advantages of this friendlier interface, the game engine will often have a community that provides different tools, models, and other helpful additions. This practice of adding functionalities is of such importance that companies will also write code so that the developer will have an easier time creating games that support their product or run on their platform. This detail was crucial in our choice of game engine and will be explained further below.

The three main engines that we considered for development were Unity, Unreal Engine, and CryEngine. We decided to use the Unity game engine for a variety of reasons:

Popular: The Unity game engine is very popular and has a large community behind it, resulting in more complete documentation, better support, and more third-party developers working on solutions to common needs.

VR Capable: While all of them can develop for VR, Unity has libraries and packages explicitly made for mobile VR development, easing the connection between in-engine testing and build testing on mobile devices.

Mobile Support: Unity is widely regarded as one of the best engines for mobile development in general, so it is no surprise that it is also beneficial for VR app development.

Friendly Interface: Unity's interface was designed with beginner game developers in mind, and thus it now benefits from a more straightforward and more intuitive user interface.

Base Simplicity: While its competitors advertise the creation of a visually stunning game with accurate physics with just a few button clicks, Unity doesn't. This may look like a fault at first. Still, on closer inspection, we concluded that not requiring high-end hardware to run a default game scene is an important asset when building for mobile devices, where processing power is not the same as desktops. We can build only what is crucial and forego any frivolous details.

Asset Store: The asset store allows us to find community-built, free, 3D models for use in our simulation, as well as libraries made to improve our code.

Google Cardboard: As we decided on using mobile devices for our application, google cardboard is one of the first methods through which mobile applications can run in VR. Google made libraries specifically for developing in unity for cardboard. We will discuss our choice of library further below.

2.6 UNITY

2.6.1 Overview

Unity is a Game Engine initially developed for macOS and launched in 2005, but now can be found on most common operating systems (Brodkin, 2013). As is the case with other engines, its objective is to aid in the development of games. Currently, Unity is written in C++ and C#. It can build on many different platforms and operating systems with minimal configuration, such as Linux, Windows, macOS, Xbox, Playstation, WebGL, and most importantly for our purposes, iOS and Android.

In figure 2.5, we can observe the main components that aid in the process of developing games with Unity. In the hierarchy view, we can see all the game objects present in the current scene and their relationship to one another. In the scene view, we can see the placement of the objects on the scene and alter them to our liking. In the inspector window, we can see the components of an object that we have selected from the hierarchy and alter the properties of these components. Finally, in the project window, we can see the assets of the project, including models, scripts, animations, sounds, and everything that we placed on the assets section of the project directory (Unity, 2021a).

Figure 2.5: The Unity Interface. 1: Hierarchy View. 2: Scene View. 3: Inspector Window. 4: Project Window.

2.6.2 Basic Concepts

To better understand how Unity works and so we can refer to these terms in the chapters to come, we will be explaining some basic concepts and ideas that are crucial to comprehend the development process (Unity, 2021b).

Material: Defines the basic elements of the object, such as texture, opacity, color, and the way it interacts with light.

Texture: Is an image that will be applied to the surface of an object. It can be stretched, tiled, and otherwise manipulated to better fit the object's dimensions.

Model: Pre-designed object that is modeled to resemble a real object, which can then be placed in the simulation.

Scene: It is the world itself and everything that exists in it at any specific time. A simpler way to understand this is to think of different game levels.

Components: Pieces of code that are attached to an object, and that can interact with the object itself or the rest of the scene. The most common component of an object in Unity is its transform, which dictates the position, rotation, and scale of the object.

Skybox: The skybox is what the user sees if it tries to look where there are no objects in the way, much like its name, the skybox is a box that surrounds the scene. Common skyboxes for daytime are drawn as a blue sky with clouds.

Scripts: Scripts are specific components attached to an object that can be easily written to add customizable functionality to the scene. More explanations on scripts and programming in Unity will be given below.

Units: In Unity, a *unit* is a measurement that is functionally equivalent to a meter. One *unit* is one meter.

2.6.3 Scripting

Scripts are where the developer can write code in C#, and Unity will interpret and execute it accordingly (Unity, 2021c). They are attached to objects as a component and can be made to interact with the scene in multiple ways. While C# is object-oriented, and thus all code is done using classes and objects, it is also event-oriented in a couple of important aspects. One of the most important of those aspects is the frame [4] update event. To render video, Unity will draw each frame on the screen sequentially, and each time the frame is updated on the screen, an event will be triggered that can be used to run any frame-sensitive code. Frames are one example of events that Unity can detect, but they aren't the only ones. Other events include UI clicks, value changes, components of objects interacting, key presses, and many more. A few reserved methods can be used to use this event functionality better, but they must be inherited from the base class *MonoBehaviour*:

Start: The start method will be executed once the script is enabled and just before any update methods are called.

Update: Update is called every frame.

Methods inherited from components: A script component will extend the methods of another of its brother components (attached to the same object), for example, a script attached to an object that has a *Rigidbody* component will be able to extend the *OnCollisionEnter* method, which will be called whenever the object collides with another object.

Because a script is a blank component, it can also have adjustable properties in the inspector window of the user interface. One classic example is using public variables in the class definition that can later be set through the interface instead of through the code. This possibility is another of the reasons why Unity is such a user-friendly game engine. Below is an example of a simple class that inherits Unity's *MonoBehaviour* class, and with it the Start and Update methods:

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class SampleController : MonoBehaviour
6  {
7        // Class Parameters creation
8        public float x;
9        public float y;
10       public float z;
11
12       // Start is called before the first frame update
13       void Start()
```

[4] Frames are each image that is rendered rapidly and sequentially displayed to make a video.

```
14          {
15
16          }
17
18          // Update is called once per frame
19          void Update()
20          {
21
22          }
23  }
```

2.7 GOOGLE VR SDK

For Unity to be able to utilize the mobile hardware natively, Google made a software development kit that could achieve this needed interaction. This Google VR SDK was what we decided to use in our application (Google, 2021c) because it was the main library for developing google cardboard apps in Unity and recommended for use by Unity themselves. At the time, the Google VR SDK was the supported development kit for building for Cardboard, but it has now become deprecated in favor of a rewritten version, called the Cardboard SDK. This happened partly because Unity decided to remove the option to build to Cardboard in their settings in favor of a new plugin made for all their XR needs. In addition, since this change was made in a more recent Unity release, we chose to continue using our version that has support for the Google VR SDK. Since Unity offers long-term support for their stable versions, we decided to continue using version 2019.4 LTS.

2.7.1 Functionality

The main functionalities of the Google VR SDK are to interpret the output of the cardboard software on the mobile device, allowing Unity to use this output and alter the application in various ways. Some of the functionalities of the SDK with access to the Cardboard software are as follows:

User Input: The user can control the simulation using the Cardboard Button.

Head Movement: The SDK uses the built-in sensor of the device to allow the users to move their heads, updating what is being seen on the screen accordingly.

Stereo Rendering: The audio has built-in stereo capabilities to create realism.

Spatial Audio: The sounds may come from different positions of the simulation to increase the immersive experience.

Unity Emulator: An emulator for testing head movement in the unity editor using a mouse and keyboard that can emulate head movement in the three degrees of freedom.

2.8 ANDROID

With our decision to use Unity to develop our application combined with the Google VR SDK and cardboard, we concluded that building to Android was a better alternative than building for iOS. While it's possible to run apps with Google Cardboard on the iOS, we believe that having less restricted access to the file system and having our own mobile devices running Android, the choice of operating system was evident. The easy access to Android's file system was of great use to our work when building the application's output and data collection systems.

An Android application package or APK is the file format of a package file that contains all the information necessary for Android to install a mobile app (Android, 2021). Because Unity will directly build an APK, we can enable the install from unknown sources option in the settings and thus easily share our APK file for testing and use without registering it in the Google Play Store.

Finally, we discovered a helpful library to add Android's Native Share functionality (yasirkula, 2021), allowing us to directly export our compiled simulation results to any location available to the operating system, most notably Google Drive and Dropbox.

2.9 CONSIDERATIONS

In this chapter, we described the main components used in our application, along with some of their functionality, essential terms, and our reasoning behind choosing to use them. The following chapter describes the software requirements raised for this project and how we implemented them with the tools described in this chapter.

3 RELATED WORKS

The general public's interest in Virtual Reality has been steadily growing over the recent years, with a substantial spike of interest recently with the rebranding of Facebook as Meta and their discourse on how the next iteration of the internet will be based on VR and AR [1] (Meta, 2021). With future newcomers to the field, there will be a demand for learning how to construct both complex and straightforward VR software in the foreseeable future. Even though the greater need for this information will ensure new content for beginners, we must not fail to account for the research and studies that have already been done.

This chapter cites studies that explore the interaction of Virtual Reality, pedestrian simulation, and the Unity game engine in different ways to learn and gain inspiration from what has been done previously or what to aspire to with more significant resources.

In the article *A Glance into Virtual Reality Development Using Unity* (ISAR, 2018), the author defines and classifies aspects of VR and focuses on identifying the most beginner-friendly game engine for VR development. They conclude that Unity is the better engine for beginners, elaborating that their use of the C# programming language over C++ offers a more friendly and easy-to-learn language. Their user interface is easy to use and explore, and they provide plenty of tutorials and excellent documentation. Furthermore, they explain that Unity can build for many platforms, which is also one of the reasons we chose it for this project, and then they set to demonstrate the ease of use of the unity engine by making a simple labyrinth game in VR.

In the article *Development of Pedestrian Simulator for the Prevention of Traffic Accidents Involving Elderly Pedestrians* (Kazutaka Mitobe and Yoshimura, 2012), the authors built a VR device to simulate street crossing in dangerous scenarios and tested it on a set of volunteers. It was created in a dedicated room with screen projectors and various sensors to work as the user's interface as seen in figure 3.1. The main goal was to evaluate elderly people's ability to avoid danger. The conclusion was that pedestrians that had a cognitive deficit would fail the test.

In the article *Distracted Pedestrians Crossing Behaviour: Application of Immersive Head Mounted Virtual Reality* (Anae Sobhani, 2017), the authors evaluated the crossing behaviors of distracted pedestrians. The visualization tool used was the game engine Unity, with the Oculus Rift [2] as HMD. They created a randomized traffic simulation to evaluate the participants. The result was that distracted pedestrians were performing more dangerous crossing compared to non-distracted pedestrians.

In the article *A Low Cost Approach to Pediatric Pedestrian Safety in Virtual Reality* (Kuldeep Pandey, 2009), the authors proposed a study on children in road scenarios, with the

[1] Augmented Reality, which allows for virtual objects to be observed in the real world
More information on AR in section 2.1.1

[2] A high-quality VR headset with screens for each eye.
The different models of headsets will be discussed in further detail in section 2.3.2.

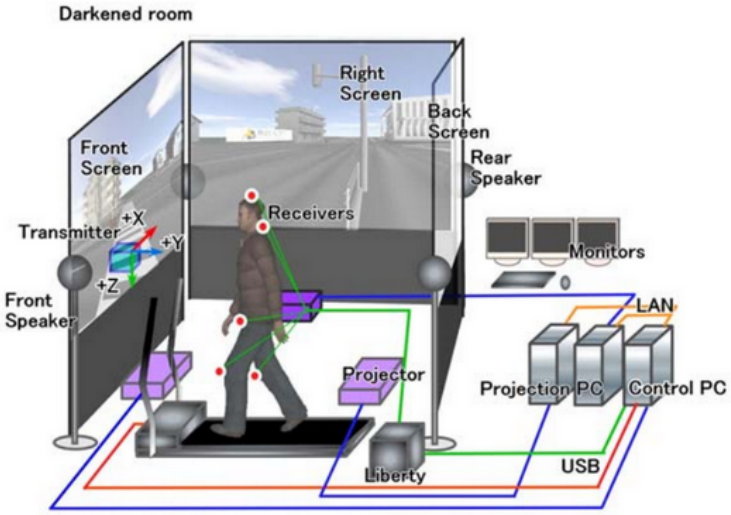


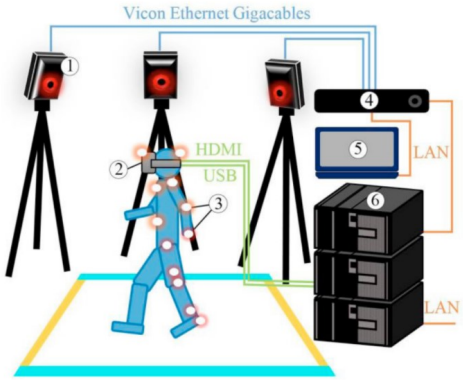
Figure 3.1: Pedestrian simulator. Reference: (Kazutaka Mitobe and Yoshimura, 2012)

main focus of measuring the behavior and training the pediatric pedestrians on safely crossing streets. They used a combination of software for the modeling and driving engine, proposing a low-cost solution. In this case, an HMD was not used; the proposed solution was a set of screens combined to simulate a 180 degrees field of view. The result was that this form of VR could be used as a tool to develop children's safety.

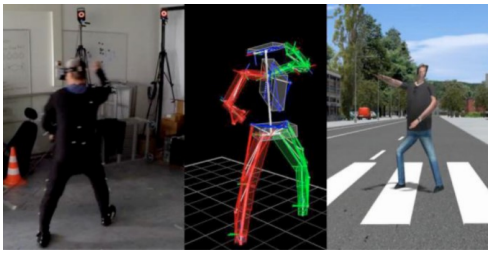
In the article *A Pedestrian Simulator for Urban Crossing Scenarios* (Feldstein et al., 2016), the authors developed a pedestrian crossing simulator in VR, with similar objectives to ours, but with more complete and immersive results. They used a combination of infrared markers to map out the subjects and their precise body movements, the Oculus Rift, and trackers to capture the spatial movement of the participants, as seen in figure 3.2. Their results included the degree to which people felt immersed in the simulation, often stretching and showing similar signs of high immersion. While indeed their research and software provide a much clearer picture of the behavior of pedestrians in a traffic situation, they do this by using expensive hardware and infrastructure, which is what we decided against for the benefits of better scalability and affordability.

3.1 CONSIDERATIONS

The works presented here served as a base for the technology and ideas used to implement our pedestrian crossing simulator in VR with Unity. In the following chapters, we will be detailing the tools and resources used to accomplish this task, along with the decisions made in favor of those tools and how we implemented our software requirements with them.



(a) Simulator Setup.



(b) Markers for tracking body movement.

Figure 3.2: Pedestrian Crossing simulator. Reference: (Feldstein et al., 2016)

4 PROPOSAL

This chapter details the software requirements that we had for this application, along with the methods and solutions we found to meet these requirements. We have grouped these software requirements into broader categories to better understand the parts necessary that when combined can result in the simulation we have envisioned. These categories are *World*, *Interaction*, *Collision*, *Log*, and *User Interface*.

4.1 WORLD

The expectation is to create a realistic world, where the player can feel as though they really might be crossing a city street. The result is partially shown in figure 4.1. The world was built respecting the scale of the real world, using Unity's *unit* as the equivalent of one real-world meter. Since everything in the simulation respects this scale, we can easily calculate distances and speeds in *units*.

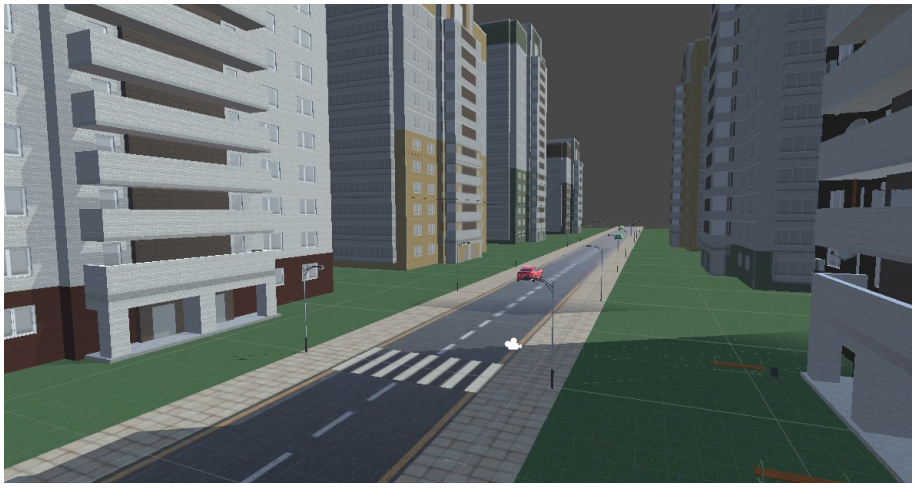


Figure 4.1: Default world scene.

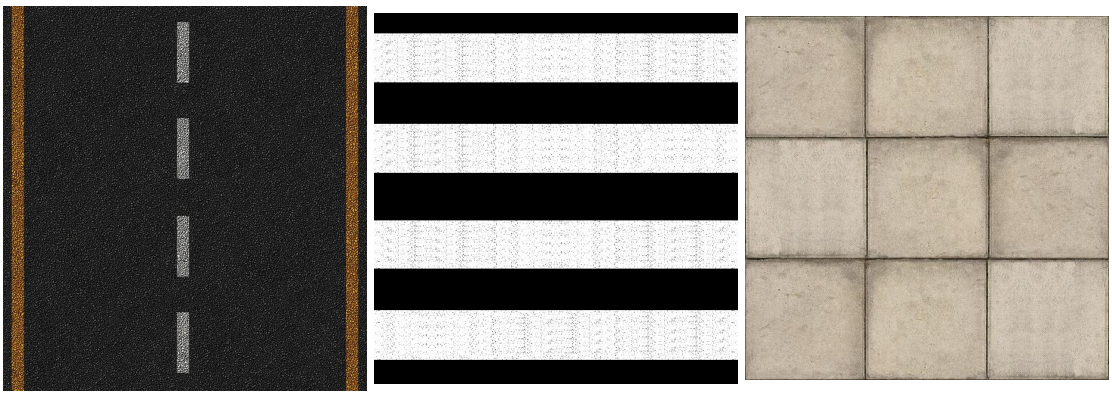
4.1.1 Street, Crosswalk, Sidewalk

The first objects added to the scene were also the most important in terms of the interaction and functionality of the simulation. The street had to have two lanes so that we could have the option of adding traffic coming from both sides or just increasing the overall flow of the street. The Sidewalk was added to be a safe space where the user can walk while deciding the moment to cross the street. The crosswalk was added on the street to measure accurate crossing in terms of location and not just timing.

The two-lane **street** was built using a plane object and a seamless textured image of a simple road to have the correct appearance. The **crosswalk** and the **sidewalk** were constructed

with the same method and were placed slightly above the street. As the object is scaled in one dimension more than the others, we must tile the texture to maintain the original image's aspect ratio. These objects were set with Brazilian streets in mind, and so their dimensions in the world are as follows:

- Street Width: 8.5 Units over the Z axis
- Street Length: 500 Units over the X axis
- Crosswalk Width: 4 Units
- Crosswalk Length: 8.5 Units
- Sidewalk Length: 500 Units
- Sidewalk Width: 3.5 Units



(a) Simple street texture. (b) Simple crosswalk texture. (c) Simple sidewalk texture.

Figure 4.2: Basic textures.

4.1.2 Surroundings and Details

The surroundings and details serve the purpose of adding to the immersion of the user, as well as blocking parts of the scene to lessen the number of objects that need to be loaded.

Buildings Buildings were added to enhance the urban feeling of the scene, with different models and colors.

Street Lamps Street lamps were added to improve the immersion further and complete the scenery, with the benefit of being a source of light when the time of day is set to night time.

Skybox Skyboxes were added to simulate the Sky view beyond what we have rendered.

World Lights Directional lights were added to create shadows and light, acting as a sun for our purposes.

(a) Building Models.

(b) Street Lamp Models.

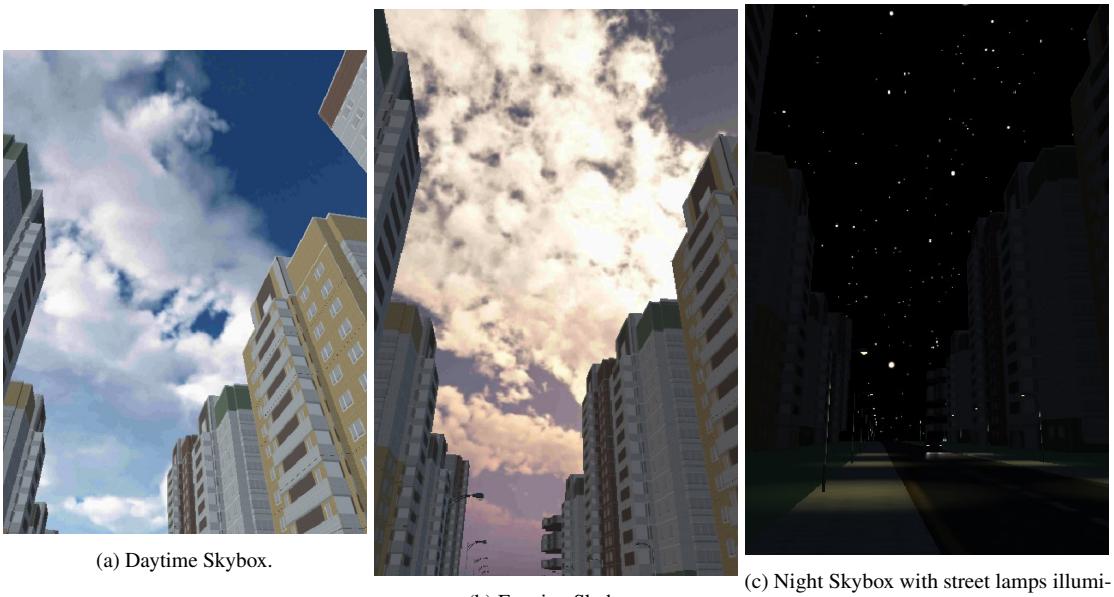
Figure 4.3: Buildings and Street Lamps add to the immersion of the scene.

The **world lights** were added by using the directional light default object and placed high above the simulation, resulting in a light source for the entire scene. The light intensity is adjusted for the time of day, resulting in night-time having almost no directional light influence.

The **buildings** were placed using nine different building models found in the Asset Store, replicated with different colors, applied to create the look-and-feel of an actual city. The building assets employ a helpful technique called the level of detail. This technique allows for game objects far away from the camera to be shown in less detail. In the case of the models used in this project, it means detailed architecture reverting to simple rectangles but resulting in less GPU load to render them.

Three different **skyboxes** were used to represent each time of day: Day with a blue sky, end of the day (sunset) with a more yellow to purple color scheme, and night with a darker color with white dots representing the stars, as seen in figure 4.4.

Street lamps were placed on the sidewalk of both sides of the road spaced by 20 units between each other. The effect of the light on the street lamps was achieved by attaching a spotlight object to the bulb and aiming it towards the floor. This effect creates an atmosphere similar to that observed in actual street lamps. The condition to turn on these lights is that the simulation is at night time.



(a) Daytime Skybox.

(b) Evening Skybox.

(c) Night Skybox with street lamps illuminating the sidewalk.

Figure 4.4: The skyboxes on different times of day with different light intensities.

4.1.3 Cars

The cars are one of the essential parts of creating the simulation, as they provide the challenge of crossing the street, the very purpose of the application. They further contribute by adding sounds to the simulation, increasing the immersion.

The car **model** used was obtained from the Asset Store, and it features a simple sedan-style chassis that can be color changed using scripts. In order to create the illusion of a constant traffic flow, a default car object was made to be later duplicated when spawning new cars. Two **headlights** were attached by using two spotlight objects on the lanterns of the model, and they can be turned on or off depending on the time of day.

The **wheels** were animated to constantly rotate as if in forward motion. This animation is necessary because the car's movement is achieved by moving the car object, and thus the wheels are not affected by friction.

Movement is achieved by showing the position of an object changing over time. Because this change in position must be displayed over several frames, it's necessary to use a frame-sensitive modifier to the translation operation of the car. This is necessary because frame rates can change in different devices, and without this modifier, the object would move faster in devices with higher frame rates. In the following code, the drive method of the car script will update the car's position on the X-axis, and when the vehicle goes out of bounds, it will destroy itself.

```
1 Drive Method:
2    Move car position in X by Speed * Delta time
3    If car position in X > Right end of the street OR
4    car position in X < Left end of the street
5       Destroy the car
```

Delta time measures the time since the last frame, and with this value, we can calculate that a speed in units per second multiplied by delta time will be the distance traveled in the time since the last frame, since delta time is a value in seconds.

Sound was added to the car by using an Audio Source component and a looped audio file that constantly plays an engine noise from the vehicle. Later, this audio source will play a braking sound instead of the looped engine noise when a collision is detected.

Traffic Control is the method that controls the rate with which new cars are generated. Depending on the new car's direction, it uses the default car template to spawn new cars at both street ends by duplicating it and changing some of its properties. Traffic control will spawn a new car every 5 seconds, alternating between lanes. The decision was made to not randomize the spawn so the simulation can be more reproducible.

The **Size** of the cars was scaled to be as authentic as possible. One of the important details is that the hitbox for the car's width was made more prominent so collisions would be detected with more accuracy. This larger hitbox accounts for the bigger space occupied by the player and partial collisions.

- Car Width: 2 Units
- Car Hitbox Width: 3 Units
- Car Length: 5.2 Units
- Car Height: 1.5 Unit
- Car Default Speed: 16.6 Units/s (60km/h)

Figure 4.5: Default car model.

4.2 INTERACTION

The User Interaction with the simulation is key to creating a proper VR environment, as described in section 2.2. In our work, the interaction is done in two major forms: the movement inside the scene providing the user the ability to walk around in the environment to choose the best place to cross and actually choosing to cross the street.

4.2.1 Player Movement

The Player movement in the scenario is limited to one axis only. The user can only walk parallel to the street; it is impossible to leave the sidewalk. This limitation was introduced to simplify the list of possible interactions for any potential users. The way player movement was implemented was very similar to car movement. The difference is that while cars will constantly be moving, players will only move while interacting with the movement keys. The movement speed for the player is 1.38 Units/s (5km/h).

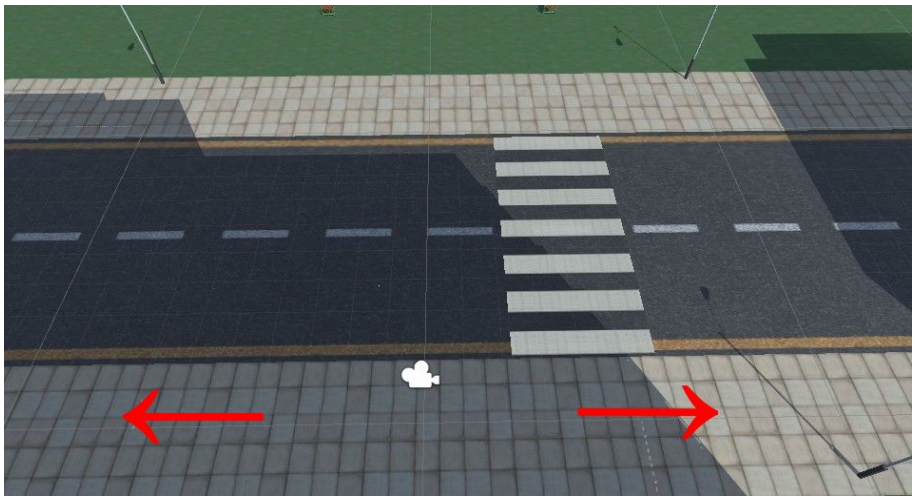


Figure 4.6: Directions of possible player movement.

4.2.2 Crossing

Because the objective of the simulation is to determine if the subject can decide on the best time and place to cross the street, the user only has control over when to start crossing the street. When the decision is made, the user will see the result of their choice before the simulation comes to an end and the collected information is given. This implementation means that the user can't suddenly decide that they chose a bad time to cross while already executing the action and can only try again on a subsequent simulation run. In this way, we ensure that the user makes a correct timing decision and can't rely on reflexes. The crossing will be done by moving the player from their current position to the other side of the street. If no collisions are detected, the simulation will end, and the end menu will be called.

4.2.3 Inputs

The Inputs are done through a **joystick controller** (figure 4.7). The User can use the analog input to set the direction of the movement based on their head position. When deciding to start crossing the street, one of the preconfigured joystick buttons must be pressed.

The joystick controller will be connected to the mobile device via Bluetooth. The joystick values can be accessed through Unity's Input Class, which handles any default input perceived by the application. Furthermore, in section 4.5, we give the user the option to set the button they prefer to use for crossing inside the options menu of the application.

Figure 4.7: An example of a bluetooth joystick controller.

4.2.4 Alternate Inputs

One of the scenarios covered in the development was the possibility of the simulation executing without a joystick controller. So an interface was created, based only on the head position, to be used as the movement and crossing input. There are three white rectangles (figure 4.8), each placed in a different position and with different functions. In order to activate the rectangle meant for crossing, the user needs to aim the crosshair directly at the rectangle for 0.5 seconds. The other two rectangles can be used by keeping the crosshair aimed at them, and they serve the function of walking left and right.

(a) Main rectangle with the crossing function.

(b) Walking rectangles, one to each side of the user.

Figure 4.8: Alternate input rectangles meant for using the simulation without a controller.

4.3 COLLISION

Collisions in the simulation happen when the player will run into the side of a car or when a car will run them over. Instead of just calculating whether the user would collide with any of the cars present in the road, we opted to give the experience of actually crossing the street and realizing at that moment why their decisions were incorrect. To achieve this immediate feedback, when a collision is detected, the player will stop, all cars in the simulation will stop, a loud braking sound will be played instead of their looped engine sounds, and the wheels will stop their animations. After three seconds of the world being still in this situation, the end menu will be called, announcing the results. Below is the collision detection method used by the simulation:

```
Collision Detection Method:
   For each car in the simulation:
      If the car's width hitbox is occupying the same space
      in the Z-axis as the player
         If the car's nose is within 1.5 units of the player
            Stop all cars and end the simulation
      If the car's nose has already passed
         If the time the player takes to hit the side of the car <
         the time it takes the car to completely move out of the way
            Stop all cars and end the simulation
```

The distance of the nose of the car to the player at which the simulation stops was set as 1.5. In testing, we found that when cars stopped too close to the player, the result of the simulation was not as impactful as seeing the car braking loudly right in front of the camera.

4.4 LOG

Collecting data on simulation executions is of the highest priority if the objective is to observe street crossing behavior. The application collects a variety of indicators that can be exported in a convenient manner and can be later studied more closely with the use of data exploration tools.

4.4.1 Collection

All information collected by the application and later exported will be listed and explained below:

- **Date:** Date of the simulation's execution.
  - **Data type:** String formatted for date (yyyy/MM/dd HH:mm).
  - **Collection Method:** Saving system time when the simulation starts.
- **Name:** Name associated with the execution.
  - **Data type:** String.
  - **Collection Method:** Text input on the options menu.
- **Age:** Age of the participant.
  - **Data type:** Integer.
  - **Collection Method:** Integer input on the options menu.
- **Height:** Height of the participant in centimeters. Will affect the height of the camera of the simulation, it is set by default to 180 centimeters.
  - **Data type:** Integer.
  - **Collection Method:** Integer input on the options menu.
- **Speed:** Speed of all the cars in the simulation in km/h. By default it's set to 60km/h.
  - **Data type:** Integer.
  - **Collection Method:** Integer slider on the options menu.
- **Alternate Controls:** Option to turn on the alternate controls when no controller is available.
  - **Data type:** String.
  - **Collection Method:** Dropdown list on the options menu.
- **Period of the Day:** Selection of the time of day the application will simulate. It can be set to day, evening, and night. This will change the light level, skybox, and whether additional lights are turned on in the world.

  - **Data type:** String.
  - **Collection Method:** Dropdown list on the options menu.

- **Difficulty:** Determines the way the cars will appear on the street. Can add the difficulty of a two way street to the simulation.
  - **Data type:** String.
  - **Collection Method:** Dropdown list on the options menu.

- **Accident:** Whether or not an accident has occurred. 0 represents no accident; 1 represents an accident.
  - **Data type:** Integer.
  - **Collection Method:** The value is set at zero by default but is changed to one if a collision is detected.

- **Accident Lane:** The lane where the accident occured. -1 if no accidents were detected; 0 if it was in the closest lane; 1 if it was in the farthest lane.
  - **Data type:** Integer.
  - **Collection Method:** The value is set at zero by default. When a collision happens, the player's location in the Z-axis will determine in which lane they are.

- **Safe Crossing:** Determines if the player has used the crosswalk to attempt crossing the street. 0 if they didn't use it; 1 if they used it.
  - **Data type:** Integer.
  - **Collection Method:** Checks if the user's position in the X-axis is within the bounds of the crosswalk when they choose to cross.

- **Crosswalk Distance:** Distance of the player to the crosswalk in units. 0 if the player is on the crosswalk.
  - **Data type:** Float.
  - **Collection Method:** Checks to see if the user is on the crosswalk. If they aren't, it calculates the distance to the nearest edge of the crosswalk.

- **Distance to the nearest car:** Distance to the nearest car that has not yet passed by the user. This can be different from the car that may have caused a collision.
  - **Data type:** Float.
  - **Collection Method:** Checks all the cars when the user decides to cross for the one with the smallest distance to the user.

- **Lane of the nearest Car:** Lane where the nearest car was found. 0 is the closest; 1 is the farthest.
  - **Data type:** Integer.
  - **Collection Method:** Checks the position in the Z-axis of the nearest car at the moment the user decides to cross.
- **Time spent deciding when to cross:** Time from the start of the simulation to when the decision to cross is made, in seconds.
  - **Data type:** Integer.
  - **Collection Method:** Timestamp of when the crossing decision was made subtracted from the timestamp at the start of the simulation.
- **Number of cars that passed by:** Counts the number of cars that passed in front of the player while they were deciding when to pass.
  - **Data type:** Integer.
  - **Collection Method:** An invisible object that has only a collider is laid out in front and on the back of the player like a giant wall (Figure 4.9). Whenever a car collides with this wall, the counter is incremented.
- **Number of Cars that passed by while crossing:** Counts the number of cars that passed in the front and back of the player while they were crossing the street.
  - **Data type:** Integer.
  - **Collection Method:** Uses the same method of the collider wall (Figure 4.9). The difference is that this counter will only increment with cars passing by while the user is crossing.

Figure 4.9: An invisible wall with a collider to count the cars that pass by the user.

- **Number of looks to the left:** Counts the number of times the user has looked to the left before crossing.

  - **Data type:** Integer.
  - **Collection Method:** Every time the user rotates their head further than 60 degrees to the left, the counter will be incremented.

- **Number of looks to the right:** Counts the number of times the user has looked to the right before crossing.
  - **Data type:** Integer.
  - **Collection Method:** Every time the user rotates their head further than 60 degrees to the right, the counter will be incremented.

- **Last left Look:** Records the time since the last time the user looked to the left before crossing in seconds. 0 if the user crossed while looking to the left; -1 if they never looked.
  - **Data type:** Integer.
  - **Collection Method:** Every time the user looks 60 degrees to the left, a timestamp will be updated. The timestamp of when the user decides to cross subtracted by this value is recorded.

- **Last right look:** Records the time since the last time the user looked to the right before crossing in seconds. 0 if the user crossed while looking to the right; -1 if they never looked.
  - **Data type:** Integer.
  - **Collection Method:** Every time the user looks 60 degrees to the right, a timestamp will be updated. The timestamp of when the user decides to cross subtracted by this value is recorded.

4.4.2 Database

As there was no need to use information stored in the database for running the simulations, the choice was made to use a JSON file stored in the mobile device's memory to keep all the information collected by the simulations. We used Unity's own *JsonUtility* package to incrementally store the results of the simulation as new elements in an array, and later to transform the JSON object into a C# object, allowing us to manipulate it when exporting the contents of our database. Below is an example of the JSON file with only one entry.

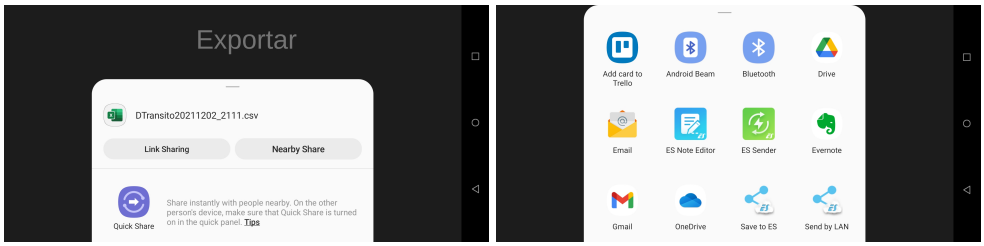
```
{
   "Table": [
      {
         "velocidade": 60,
         "cruzamentoCorreto": 0,
```

```
            "periodo": "Dia",
            "controleAlternativo": "Sim",
            "dificuldade": "Mão Dupla",
            "nome": "Nome Vazio",
            "idade": 0,
            "altura": 180,
            "timestamp": "2021/12/02 16:20",
            "distanciaCruzamento": "2.09",
            "distanciaCarroMaisProximo": "57.13",
            "faixaCarroMaisProximo": 0,
            "tempoParaTomadaDeDecisao": 2,
            "quantidadeDeCarrosQueJaPassaram": 0,
            "quantidadeDeCarrosEnquantoAtravessava": 2,
            "faixaAcidente": 1,
            "houveAcidente": 1,
            "quantidadeDeOlhadasEsquerda": 0,
            "quantidadeDeOlhadasDireita": 0,
            "ultimaOlhadaEsquerda": 2,
            "ultimaOlhadaDireita": 2
        }
    ]
}
```


4.4.3 Exporting

In order to export the information kept in our database into a more readable format, the choice was made to use the *CSV* file format. Almost any software can read the CSV format, especially those developed for analyzing data. Transforming our JSON file into a CSV output file, also stored in the mobile device, was done by reading everything in the JSON file and converting it back into an object with an array that has all the lines as elements. With this array in hands, the only thing left to do was to iterate over its values and store them as comma-separated values in a new file. As CSV files are so compact, and it is almost impossible to produce the amount of data that would cause the size of our files to become a problem, the decision was made to give the users only two options when exporting: export the entire database or export the results obtained at the current date, as a convenience.

With the files being successfully exported, the next step was to have a more convenient way of accessing them from the mobile device's memory and possibly store them in an adequate location. For this purpose, a community-built library for Unity was found, *Native Share* (yasirkula, 2021). Native share gives Unity the tools needed to interact with Android's native file-sharing tools, and with this, the ability to prompt the user with Android's system prompt asking where to share the file. Since Android can share files to Google Drive, Dropbox, Messaging apps, E-mail, and similar services, our application can use them to save our exported files in a more convenient location.



(a) CSV file shared with Android. (b) Multiple apps to share the file.

Figure 4.10: Native Share using Android to natively share our exported file.

4.5 USER INTERFACE

Since interacting with UI elements in VR is difficult, the decision to keep all of the navigable menus without any form of stereoscopy was made. This choice allows the usage flow of the application to start with the device in hand, adjust all the settings necessary with the precision of fingers, start the simulation, experience the street crossing simulation, return to navigable menus to export the collected data or run the simulation again.

4.5.1 Main menu

The main menu is the first scene presented to the user when starting the application. Once inside, the user can access the simulation immediately, enter the options menu or the export menu.

Figure 4.11: Main menu with buttons in order: Start, Options, Export.

4.5.2 Options

The options menu has all the settings that the user can change to customize the simulation.

Name Input: The user can insert their name to be registered in the execution logs.

Age Input: The user can insert their age to be registered in the execution logs.

Height Input: The user can insert their height in centimeters, which will impact the game height of the camera in the simulation.

Speed Slider: Slider to define the speed of all cars inside the simulation.

Period of the Day: A dropdown option to choose between: day, evening, and night, inside the simulation.

Alternate Controller: A dropdown to enable or disable the alternate controls.

Difficulty: A dropdown option to choose between a one-way or a two-way street.

Back Button: Goes back to the Main Menu.

Controller: Enters the Controller Menu.

Figure 4.12: Options menu with all configurable settings.

4.5.3 Controller

The controller menu can be used to keybind the crossing button to whatever button the user prefers on their controller. After the UI button is pressed, the next joystick button detected by the application will be set as the crossing button, and the pressed button will be shown in text form so the user can be sure they pressed the correct button.

Figure 4.13: Controller menu with the keybind and main menu buttons.

4.5.4 Export

The export menu will use Android's native share functionality to conveniently share the created CSV file with either all collected data or the collected data of the current day.

Figure 4.14: Export menu with buttons in order: Export everything, Export from today, Main menu.

4.5.5 End Menu

The end menu will confirm whether the user was able to cross the street without accident or not. It is displayed immediately after the simulation ends.

Figure 4.15: End menu showing the overall result of the current simulation, along with the Main menu and Export buttons.

4.6 CONSIDERATIONS

This chapter highlighted all the software requirements needed to create the simulation, like modeling the world and its details, coding the interactions between the objects and the user, defining the way collisions would be detected and represented, logging relevant information for further study and analysis, and designing a friendly UI that can meet our configuration needs. In the next chapter, we will test and validate the application in different ways to display its possibilities for potential users.

5 EXPERIMENTS AND RESULTS

This chapter presents qualitative tests of the application, with the goal of verifying the proper functioning of the requirements proposed for the development.

5.1 VALIDATION

The validation was built based on individual executions of scripted use cases to verify each output behavior for each case described below. As the tests were done in order to validate that the application works as expected, validating the application for use in psychological or other research was not done. This should be kept in mind for future uses of the simulation.

5.1.1 Stereoscopic display

The application must properly display stereoscopic images when building on Android. The screen was captured while executing the simulation to verify this ability in a situation where the difference in perspective between the images can be clearly seen. In figure 5.1, a building on the nearest side of the street can be seen on the left edge of the left image, while that same building is not present on the left edge of the right image. Furthermore, the pink car shown on the right side of both images can barely be seen on the left image, but the entire wheel and back window can be observed on the right image.

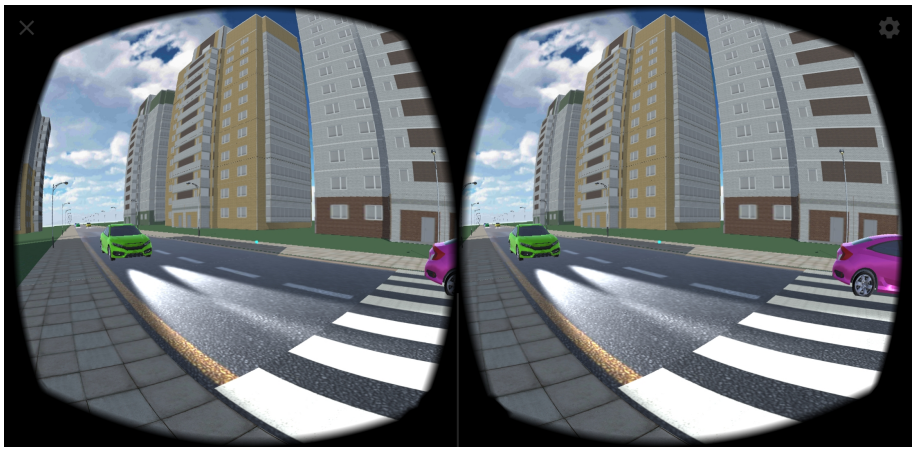


Figure 5.1: Captured simulation screen with differences between the images shown to each eye.

This validates that the application is rendering slightly displaced images based on each eye to create the illusion of depth in the mobile device's screen.

5.1.2 Log collection

Validating that each field in the log file represents the user's actions is essential to ensuring the collection of data is correct. The use cases made for guiding a subject through the simulation in a scripted manner should return logs that correctly reflect their instructions. The use cases were the following:

In table 5.1 it can be seen that only four fields in the results didn't match. Because "Distance to the nearest car" can't be observed and followed precisely by the subject when executing these guided simulations, the results are expectedly different. The other three fields that didn't match were: "Time spent deciding when to cross," "Last left Look," and "Last right look." All three are time-related, and the validation needs a human to interact with an external handheld chronometer. The maximum time difference was 0.5 seconds, which could be considered an expected human error difference.

5.1.3 Speeds and Time

In order to validate if the speeds were correctly implemented, two methods were used. The first involved using the *Log* function with timestamps to calculate if the speeds were appropriate while not relying on the time calculations employed by the application's own code. This was achieved by logging the spawn and destroy times of newly generated cars. Since new cars have to travel the entire length of the street before being destroyed, that gives us a distance of 500 Units. This test was done using the default speed of 60km/h or 16.6 units/s. If the distance to be traveled is divided by the speed, this gives us the time to travel the distance, which in this case is 30 seconds. Below we can see car 26 spawning at 32:50 and being destroyed 30 seconds later, at 33:20.

```
[14:32:50] Car 26 is spawned.
UnityEngine.Debug:Log (object)
Car:Start () (at Assets/Scripts/Car.cs:18)

[14:33:20] Car 26 is destroyed.
UnityEngine.Debug:Log (object)
Car:Dirigir () (at Assets/Scripts/Car.cs:34)
Car:Update () (at Assets/Scripts/Car.cs:25)
```

Another important speed to be validated is the player's speed, which was set to be 5km/h or 1.38 units/s. The distance needed to cross the street is a little larger than the width of the street, due to walking from sidewalk to sidewalk and not the edges of the road. The time must be recorded from the moment the player presses the cross button to when they stop moving after crossing the street. The distance traveled will be ten units from sidewalk to sidewalk. Ten units divided by the speed of 1.38 units/s gives us 7.24 seconds. The time recorded by the handheld chronometer used was of 7.49 seconds, a difference of 250 milliseconds that can be attributed to a human reaction time error. With this, the speeds in the simulation can be trusted and used in further studies.

5.1.4 Replicability

Validating that the world is generated in the same way was done by using time as an indicator of when to pass and receiving the same results. Using a handheld chronometer to ensure the correct timing was done by choosing a number of seconds, in this case, three, to wait before pressing the button to cross the street in two different simulation executions. This validation was done for each difficulty (One-way and two-way streets). Below is the comparison between the results obtained in each difficulty.

Table 5.2: Comparison of results between two One-way difficulty tests and two Two-way difficulty tests.

	One-way(1)	One-way(2)	Two-way(1)	Two-way(2)
`Time deciding`	3s	3s	3s	3s
`Accident`	Yes	Yes	Yes	Yes
`Accident Lane`	Closest	Closest	Closest	Closest
`Correct Crossing`	No	No	No	No
`Crossing Distance`	2.09 Units	2.09 Units	2.09 Units	2.09 Units
`Nearest Car`	16.69 Units	25.67 Units	39.43 Units	33.18 Units
`Nearest Car Lane`	Farthest	Farthest	Closest	Closest
`Speed`	60km/h	60km/h	60km/h	60km/h
`Cars passed deciding`	0	0	0	0
`Cars passed crossing`	1	1	0	0

In table 5.2 we can see differences in the results between the tests. In the one-way difficulty tests, the nearest car is closer to the player in the first test by 9.28 units. With the car speed at 60km/h or 16.6 units/s, we can calculate that this distance difference was because of approximately 500 milliseconds of delay when pressing the crossing button between the tests. This is accounted for by human error when repeatedly using an external tool such as a chronometer. The same happens in the two-way difficulty tests.

Another difference is that in the one-way tests, the nearest car is on the farthest side of the street while the accident happened in the closest lane. This happens because there is a car ahead of the one the player collides with, but it passes by in the farthest lane while the user is still trying to cross the first lane. We can see evidence of this in the last line of the table, where a car was recorded as passing by the user while they are already crossing.

5.1.5 Input

In order to validate the input, three volunteers were requested to play the simulation using the controller and the alternate controls. The result was that all three could use the movement commands, going to the left and right, and the crossing command was used to cross the street. This result shows that the Interaction with the game matches the expected behavior.

5.2 CONSIDERATIONS

In this chapter tests were presented to validate the functionality of the application in accordance with the proposal. For each test, the results were analyzed and justified according to the original motivation of the simulation.

The next chapter will discuss the development and test results, possible uses for this piece of software in other fields of study, and ways to improve the simulation in further iterations.

Table 5.1: Log fields validation

	Use Case	Output Expected	Output Log
`Date`	Execute the game and verify if the Timestamp Matches with an external clock	2021/11/10 17:32	2021/11/10 17:32
`Name`	On the Option Menu, insert "João" in the Name field	João	João
`Age`	On the Option Menu, insert "99" in the Age field	99	99
`Height`	On the Option Menu, insert "100" in the Height	100	100
`Speed`	On the Option Menu, select 40km/h in the Speed Slider	40	40
`Alternate Controls`	On the Option Menu, select "Sim" in the Alternate Control selector	"Sim"	"Sim"
`Period of the Day`	On the Option Menu, select "Noite" in the Period of the Day dropdown	"Noite"	"Noite
`Difficulty`	On the Option Menu, select "Mão Dupla" in the Difficulty dropdown	"Mão Dupla"	"Mão Dupla"
`Accident`	Create an accident when crossing the street	1	1
`Accident Lane`	Create an accident on the first lane	0	0
`Safe Crossing`	Move the position to be in front of the Crosswalk and make the cross	1	1
`Crosswalk Distance`	Move the position to be in front of the Crosswalk and make the cross	0	0
`Distance to the nearest car`	Not Able to test		
`Lane of the nearest Car`	Cross the Street when the nearest car is on the first lane	0	0
`Time spent deciding when to cross`	Start a handheld chronometer when the game starts and when it comes to 5 seconds, make the cross	5	5.4
`Number of cars that passed by`	Count the numbers of cars that passed by, when it comes to 13, make the cross	13	13
`Number of Cars that passed by while crossing`	Make the cross when a car on the second lane is close to you, but will not cause an accident	1	1
`Number of looks to the left`	Looks 3 times to right and 1 to left then make the cross	1	1
`Number of looks to the right`	Looks 3 times to the right and 1 to the left	3	3
`Last left Look`	Look to the left, start a handheld chronometer at the moment you look to the front, after 5 seconds on the handheld chronometer make the cross	5	5.5
`Last right look`	Look to the right, start a handheld chronometer at the moment you look to the front, after 6 seconds on handheld chronometer make the cross	6	6.4

6 DISCUSSION AND CONCLUSION

This chapter discusses some of our experience developing this project, along with possible uses and possible future improvements.

6.1 CONSIDERATIONS

In this work, we have developed a street crossing simulator for android, built using unity, and using google cardboard software to enable VR functionality on a mobile device. This application can be adjusted in various ways to more appropriately simulate the scenario being studied.

In testing, the simulation was found to be of sufficient quality to induce a truly immersive virtual reality experience in the subjects. With the various scales, speeds, and data collection systems being validated, we have observed that while not the most realistic and immersive solution possible to this field of study, it will be advantageous especially in regards to affordability, ease of use, and portability. The immersion achieved using a mobile device was found to be more than satisfactory in relation to the prices involved, and just this detail can already make possible many more uses for such a simulation in the future.

After validation, testing, and considering the aggregated costs, we could justify suggesting this application for use in future studies of pedestrian behavior in traffic, after further studies validating this simulation as a research tool, and conclude that this work fulfills our original objectives and motivations.

6.2 POSSIBLE USES

The most important possible use is to enable previously impossible studies, either because of price, portability, or other factors to become possible. This is especially relevant for research done by students or other researchers with not enough funding for dedicated VR hardware and software.

Because the application was made as a data-gathering tool, it doesn't restrict the user in terms of its applicability. This means that it can be used for many purposes, notably studying and understanding human psychology and behavior when in a traffic situation, after further testing and validation for this purpose.

The nighttime setting makes for a very unique experience in terms of gathering information, as most traffic simulators will usually be well illuminated and have very visible cars. Being guided mostly by headlights and streetlamps can be another aspect and research avenue to be explored in the future.

We suspect that collecting large amounts of data on the behavior of users in the one-way street mode might reveal that people do not take the adage of always looking at both sides before

crossing the street seriously enough, as the risk of collisions with vehicles driving on the wrong direction of the street is always prevalent. The same can be said for using the crosswalk and many other factors that may be discovered with gathered data.

6.3 FUTURE WORKS

Having the results and experience gathered in the development process, we can see possible future works and improvements that can be done to enhance the application.

6.3.1 Simulation

- Traffic control option that uses real traffic models to simulate car creation and behavior
- Variable speeds for each car, with interactions between them.
- Saving information on more cars on each simulation.
- Tutorial scenes to teach the proper way of crossing the street.
- Adding more scenarios, such as rural, near a school, and others.
- Variable speeds, for walking and running, maybe taking height into account.
- Different traffic interactions, like traffic signals and signs that can be interacted with and will alter the behavior of the simulation.
- Adding a body that the user can see to enhance the immersion.

6.3.2 User Interface

- Saving multiple configuration settings for easier repeated use of the simulation.
- Adding the possibility to configure the simulation settings on a different device. This way, the researcher can easily configure or change any setting without removing the device from the subject's head.
- More options for the scale of the simulation, including bigger cars, streets, buildings, and other elements.
- Options to add lanes, streets, crossroads, and other traffic settings.

6.3.3 Hardware

- Making a PC version that supports the high-end HMDs of the market.
- Creating an Iphone version of the application.

REFERENCES

Anae Sobhani, Bilal Farooq, Z. Z. (2017). Distracted pedestrians crossing behaviour: Application of immersive head mounted virtual reality. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–6, Yokohama, Japan.

Android (2021). Application fundamentals. https://developer.android.com/guide/components/fundamentals. Visited on 2021/12/03.

Balasubramanian, S. (2021). The next frontier for healthcare: Augmented reality, virtual reality, and the metaverse. https://www.forbes.com/sites/saibala/2021/11/29/the-next-frontier-for-healthcare-augmented-reality-virtual-reality-and-the-metaverse/. Visited on 2021/12/03.

Bardi, J. (2021). What is virtual reality? [definition and examples]. https://www.marxentlabs.com/what-is-virtual-reality/. Visited on 2021/12/03.

Brodkin, J. (2013). How unity3d became a game-development beast. https://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/. Visited on 2021/12/03.

Feldstein, I., Dietrich, A., Milinkovic, S., and Bengler, K. (2016). A pedestrian simulator for urban crossing scenarios. *IFAC-PapersOnLine*, 49(19):239–244.

Florian (2018). Best game engines for beginners: 14 alternatives. https://www.tooltester.com/en/blog/best-game-engine/. Visited on 2021/12/03.

G1 (2013). Driving simulator rig. http://g1.globo.com/to/tocantins/noticia/2013/09/simulador-de-direcao-e-retirado-de-exposicao-por-problemas-tecnicos.html. Visited on 2021/11/29.

Goode, L. (2019). Get ready to hear a lot more about 'xr'. https://www.wired.com/story/what-is-xr/. Visited on 2021/12/03.

Google (2021a). Augmented reality. https://arvr.google.com/ar/. Visited on 2021/12/03.

Google (2021b). Google cardboard viewer. https://arvr.google.com/cardboard/. Visited on 2021/11/29.

Google (2021c). Google developers / google vr. https://developers.google.com/vr/develop. Visited on 2021/12/01.

Heilig, M. L. (1962). Sensorama simulator. https://patents.google.com/patent/US3050870A/en.

ISAR, C. (2018). A glance into virtual reality development using unity. *Informatica Economică*, 22(3/2018):14.

Kazutaka Mitobe, M. S. and Yoshimura, N. (2012). Development of pedestrian simulator for the prevention of traffic accidents involving elderly pedestrians. In *SICE Annual Conference 2012*, pages 1365–1368, Akita University, Akita, Japan.

Klein, A. (2021). The world of 3d-imaging,. https://www.stereoscopy.com/. Visited on 2021/12/03.

Kuldeep Pandey, G. J. G. (2009). A low cost approach to pediatric pedestrian safety in virtual reality. In *2009 Third Asia International Conference on Modelling & Simulation*, pages 549–554, Bundang, Indonesia.

Lemon, M. (2013). Vr omni-directional treadmill lets players run and gun. http://www.escapistmagazine.com/VR-Omni-Directional-Treadmill-Lets-Players-Run-and-Gun/?utm_source=rss&utm_medium=rss&utm_campaign=news. Visited on 2021/12/03.

Meta (2021). Introducing meta: A social technology company. https://about.fb.com/news/2021/10/facebook-company-is-now-meta/. Visited on 2021/10/28.

MI/CNT (2019). ResoluÇÃo nº 778, de 13 de junho de 2019. *DIÁRIO OFICIAL DA UNIÃO*, 115(1):25. Visited on 2021/11/09.

Milgram, P., Takemura, H., Utsumi, A., and Kishino, F. (1994). Augmented reality: A class of displays on the reality-virtuality continuum. *Telemanipulator and Telepresence Technologies*, 2351.

Oculus (2021). Oculus quest. https://www.oculus.com/. Visited on 2021/11/30.

Qualcomm (2017). On-device motion tracking for immersive mobile vr. https://www.qualcomm.com/media/documents/files/on-device-motion-tracking-for-immersive-vr.pdf. Visited on 2021/11/30.

R7, c. R. T. (2021). Cresce o número de multas por desrespeito à faixa de pedestre. https://noticias.r7.com/brasilia/cresce-o-numero-de-multas-por-desrespeito-a-faixa-de-pedestre-03092021. Visited on 2021/11/08.

Sony (2021). Playstation vr. https://playstation.com/. Visited on 2021/11/30.

Statt, N. (2014). Facebook has oculus, google has cardboard. https://www.cnet.com/news/facebook-has-oculus-google-has-cardboard/. Visited on 2021/12/03.

Sutherland, I. E. (1968). A head-mounted three dimensional display. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, page 757–764, New York, NY, USA.

Torquato Steinbakk, R. and Bianchi, A. (2010). Comportamento de risco do pedestre ao atravessar a rua: Um estudo com universitarios. *Transporte: Teoria e Aplicaçao*, 2:19–41.

Unity (2021a). Unity. https://unity.com/. Visited on 2021/12/01.

Unity (2021b). Unity documentation. https://docs.unity.com/. Visited on 2021/12/03.

Unity (2021c). Unity scripting. https://docs.unity3d.com/Manual/ScriptingSection.html. Visited on 2021/12/03.

Vive (2021). Htc vive pro. https://www.vive.com/. Visited on 2021/11/30.

Weiss, L. C. G. (2019). Desenvolvimento e comportamento de crianças pedestres. Master's thesis, Pós-Graduação em Psicologia, Setor de Ciências Humanas, Universidade Federal do Paraná, Curitiba - PR.

Wirtz, B. (2021). How to make your own video game engine. https://www.gamedesigning.org/learn/make-a-game-engine/. Visited on 2021/12/03.

yasirkula (2021). Native share for android & ios. https://assetstore.unity.com/packages/tools/integration/native-share-for-android-ios-112731. Visited on 2021/12/02.