POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Realistic path development in a virtual environment for detection and rehabilitation treatments of Alzheimer's disease patients

Supervisors

Prof. Vito DE FEO

Prof. Andrea SANNA

Candidate

Daniele BIGAGLI

July 2022

Summary

Today, virtual reality is experiencing significant progress with respect to its technologies, and although it struggles to find a place in the more commercial and consumer environment, from a professional perspective it emerges in applications and functionalities. With increased performance and lower costs, virtual reality devices are being used for staff training, remote intervention, meetings in virtual environments, learning and health simulations. In particular, virtual reality is increasingly being used for medical analysis, detection and rehabilitation, being able to assist in interventions of patients with physical or cognitive impairments.

It is with these objectives in mind that PEDAL project was born, which aims to create an explorable virtual environment to allow immersion for patients with cognitive difficulties or diseases such as Alzheimer's. The idea is to act directly on the training of memory and spatial orientation, predominant factors in the context of diseases of this kind. By combining a virtual reality device with a stationary bicycle with handlebars, the intention is to let the patient move within the virtual environment by simulating a bicycle ride, trying to orient himself within a city district by repeating predetermined paths.

In this thesis, the problem of defining orientation paths is examined, aiming to create a tool that allows rapid and scalable definition of paths, executable by bicycle within a given virtual environment. The first objective of the work is to create an abstract representation of the environment, which can be used by subsequent search algorithms as well as other accessory components, and use it to identify paths between two selected points that meet the requirements of practicability, length and realism.

It is then ensured that each identified path has all the parameters and components needed to implement path learning phases, using indications, symbols or textual descriptions, and performance evaluation, so as to have a track of the progress made in several treatments. In order to support the user during the various phases of the application, a virtual assistant is introduced that can function as a guide and as the main source of interaction for the user, and that is capable of reacting according to the patient's progress with respect to the path to be followed. Finally, it is set the task of introducing a management system of pedestrians NPCs that move in

certain areas of the map, defined by the environment representation algorithms themselves.

For the project's development, Unity, one of the most popular and supported game engines, was used, constructing a city environment from a reproduction of a section of the city of Colchester, combining the virtual scene system with the scripting mechanism it implements through the C# language.

Referring to numerous path planning researches in the field of robotics and video games, various path finding algorithms were analysed, in particular those related to A*, to define an algorithm that would search for the best path to get from one point to another on the virtual map and that would meet the requirements imposed. A tool was implemented for Unity, which refers to the game engine's Editor library, which allows the representation of the virtual environment to be extracted from the 3D model, analysing the presence of roads and buildings, and on which, providing a starting point and destination, the defined path finding algorithms can be applied, using elements of the Unity user interface.

The virtual assistant, imagined as a small floating robot and created by a member of the project's graphics and rendering team, was integrated into the virtual environment so that it could refer to the abstract representation, by nodes and connections, of the city. Functionality has been added to it that allows it to obtain information on the user's status, such as a track of the most recently travelled road, the direction taken and the stop times, and consequently be able to process it to give advice or directions to the user, using 3D interface systems, animations and particle effects.

Finally, by using 3D humanoid models as placeholders, a system was added for the appearance and disappearance of pedestrians moving through areas identified as walkable, relying on Unity's coroutine mechanism.

In order to be able to refine the parameters that constitute the steps of the path finding algorithm used, a small virtual urban area with streets, buildings and pavements was created. In this environment, multiple applications of path finding algorithms were performed until an initial configuration of the parameters was found to be satisfactory.

Starting from these results, the Turn, Bounds and Orientation Sensitive A* algorithm was applied to the virtual Colchester to test the outcomes and further refine parameters, arriving at the final configuration used for path definition.

Finally, up to 5 paths were defined using implemented methods and tested by 5 different subjects who tried out the platform on a PC, controlling their movements with mouse and keyboard. For each path, the length in metres and the average travel time of the entire route were pre-calculated. In the tests, it is checked whether the distance covered by the subjects and the time taken to follow the path, knowing its directions, come close to the pre-calculated values, proving that these may be optimal values as parameters for the evaluation phases. It is also tested

whether the behaviour of a subject moving on the streets of the virtual city is similar to the expected behaviour described by the identified path.

This document is divided into six chapters; the first chapter introduces virtual reality and how it is applied in the field of healthcare. The PEDAL project is then briefly presented. In the second chapter, a number of articles and studies concerning treatment and rehabilitation with virtual reality are illustrated. This is followed by a description of the topic of pathfinding, and then a number of articles on path search in real-life situations and in video games are presented. In the third chapter, the hardware technologies involved in the PEDAL project are described, as well as software technologies, in particular by delving into Unity. The objectives of this thesis are described in depth in the fourth chapter, following with full details of the implementation and study of the involved algorithms. In the fifth chapter, some applications of the pathfinding algorithms are presented to show the most important differences in their results, followed by the outcomes of some motion tests in the platform to make evaluations with respect to parameters characterising the defined paths. Finally, the sixth chapter presents the conclusions.

Acknowledgements

Thanks to my family for always supporting me in pursuing my dreams and goals until now. Thanks also to all the friends and people who have accompanied me on this path. Finally, thanks also to all the people involved in the PEDAL project for their mutual help and cooperation, it was a pleasure to take part in this experience.

Table of Contents

List of Tables

List of Figures

Acronyms

AD Alzheimer's Disease

AI Artificial Intelligence

DPA Dynamic Pathfinding Algorithm

HMD Head-Mounted Display

IDA* Iterative Deepening A* algorithm

IDE Integrated Development Environment

LOS A* Line of Sight A* algorithm

MCI Mild Cognitive Impairments

NPC Non-Playable Character

PEDAL Personalized Environment for Dementia Assisted Living

SDK Software Development Kit

TOBS A* Turns, Orientation and Bounds distance Sensitive A* algorithm

UI User Interface

VA Virtual Assistant

VR Virtual Reality

Chapter 1

Introduction

1.1 Virtual Reality

Virtual reality has always been described in several ways, and looking at its most recent definitions, it can be found that VR is a "real-time interactive graphics with 3D models, combined with a display technology that gives the user the immersion in the model world and direct manipulation" [1], or it refers to "a immersive, interactive, multi-sensory, viewer-centered, 3D computer generated environments and the combination of technologies required building environments" [2]. Althought slightly different, these definitions and most of the others share key features that are defining virtual reality as it is; the simulation of an immersive experience in a reconstructed virtual environment, which people can perceive with multiple senses, and where they can interact and have an impact on things that compose this environment.

There are three types of VR systems, depending on the level of the immersion they provide:

- Fully-Immersive VR is the most realistic virtual experience; it encases the visual and audio perception of the user in the virtual world, cutting out all that belongs to the reality. [3] This system is often realized with Head-Mounted Display that tracks user's head movements and gloves to let them touch and interact with objects inside the virtual environment. This technology makes the user feel part of the recreated world, as if events are happening to themselves. Walking through cities or driving virtual vehicles are examples of an immersive virtual reality.

- Non-Immersive VR is a system realized with common displays and applications that resemble something like a window to a virtual world; while providing a lower level of presence and interaction, it could achieve an increase comfort

and a lower cost overall. This technology provides a computer-generated environment but lets the user have full control over their movements and the physical world around them. Inputs such as keyboards or controllers are used. Classic video games are generally an example of non-immersive virtual reality.

- Semi-Immersive VR is somewhat of an hybrid of the previous systems, where developers combine the simplicity of the VR desktop with a higher level of immersion, using additional devices such as gloves, joysticks, glasses. This kind of technology is often used for educational and training purposes, for instance flight simulator systems.

Common recent devices used for immersive Virtual Reality are gaming HMD such as Oculus Rift, HTC Vive, Playstation VR. Oculus and HTC have developed and distributed standalone headset; in exchange for a lower performance, they do not need to rely on an external support system such as a PC or a gaming console.

1.2 VR Technology for healthcare

Virtual Reality technology has been evolving rapidly for several years, with ever-increasing performance growth, while at the same time becoming more accessible in terms of cost and usability. The employment of VR in analysis and healthcare is therefore rising, where ever more companies have adopted these tools for the medical professionals' training, education, and interventions or treatment of patients suffering from various diseases. VR technology has the possibility of simulating environments, situations, points of view that are normally impossible to reproduce, granting a chance to other people to experience a pathological condition that could be difficult to explain or imagine, allowing to contribute to an improvement in the treatments dedicated to these pathologies.

In general, rehabilitation in healthcare refers to the treatment and process to restore good health and regain impaired functions and abilities [4]. Two main categories can describe different types of impaired functions: physical impairments are the ones related to injuries and difficulties on movement and control of body parts such as hands, legs, head, limbs, while cognitive impairments are instead related to complications in vision, hearing, memory. The latter are the most challanging cases, with diseases that affect brain and neuron system directly

Virtual Reality has had a major impact in the field of mental health; over the years, applications have been developed that allow therapies for the treatment of anxiety, the improvement of the attention deficit, recovery from post-traumatic stress. It is also used for the treatment of kind of phobias such as the fear of driving, fear of flying, fear of specific animals and insects [5]; since therapies of this type are managed through the principle of exposure, with VR it's possible

to live situations that are impractical or impossible to recreate, reducing the cost widely and minimizing the risks. The feeling of being in a fully controlled virtual environment, which can be switched off if necessary, helps the patient to feel safe and more efficient [6]. Other important fields of applications of the Virtual Reality are diagnosis and rehabilitation treatments for patients with stroke recoveries, schizophrenia and dementia.

Moreover, advantages of the virtual reality are given by the full control over stimulus presentation and response measurement, according to medical operators or researchers [7]. It's possible for developers and rehabilitation specialists to design virtual environments and simulations to adapt the system to requirements based on patient's status and impairments. VR systems are often flexible to gradual differentiations in difficulty and challenge of specific sessions, allowing, when possible for the patient, independent treatments performed individually, with non-medical support, or remote.

1.3 PEDAL project

The Personalized Environment for Dementia Assisted Living project, PEDAL, aims to become an accessible support for the treatment of patients with Alzheimer Disease (AD) or Mild Cognitive Impairment (MCI), combining a common VR headset with a stationary bicycle. Patients can navigate a digital realistic neighborhood while riding the bike and be stimulated to find their way back to a specific destination defined in the map. The stationary bike, connected to a smart trainer, allows a real movement of the patient in a confined space, which could be the therapeutic office or a dedicated room in a house. This treatment can positively improve those aspects that are generally affected by diseases such as Alzheimer's, not only short-term and long-term memory, but also orientation as well as the ability to learn, speak and move efficiently; in a PEDAL session, the patient is asked to memorize and reproduce the shortest route to return to their virtual home, testing their memory and their spatial orientation skills.

With the guidance of coordinator Prof. Vito De Feo, the work of this thesis consisted of implementing various artificial intelligence related components. First, multiple tools and algorithms were implemented that allow developers to summarise a virtual environment into a graph of walkable or obstructed nodes, which can be used to perform subsequent path search investigations on them. With this in mind, the research started from the basics of the Dijkstra algorithm to explore other variant algorithms and to test them for different goals; path defining, indications, direction tracking at runtime. Using this graph system, the logic behind the virtual assistant was implemented; a robot character, created by the rendering and animation team, with the aim of teaching the user how to travel the path to a

certain destination, giving indications and hints. In the end, other Non-Playable Characters have been added to the environment, to simulate pedestrians that populate a city area.

Chapter 2

Related Works

2.1 State of the Art

The efficiency of VR technology for supporting people with Alzheimer's or Mild Cognitive Impairment [8] has been a hotly debated issue over the years. There are numerous studies about it, which are mainly divided into two different fields: tests that deal with discriminating, among a larger group of people, those who are affected by AD or dementia from the controls, putting all the examined in conditions to try to perform some kind of tasks in a virtual environment, and longer-term treatments where patients are invited to repeat training sessions several times in order to verify any positive results in preventing a decay of neurological conditions or even improving their state.

In earlier times, studies such as [9] were illustrations of a typical research in this area. In this study, it has been prepared a computer-simulated virtual environment to assess daily living skills in a sample of people with traumatic brain injuries (TBI). A group of people capable of independent living, but with medical diagnosis of closed brain injury, were tested with a simulation where patients had to interact with a reproduced kitchen environment, where they would have to prepare a can of soup. Results of this research have stated that task executed in familiar environment have the potential to be an assessment and training support for these people.

In 2012, researchers tried to understand which aspects of the VR rehabilitation systems affect better recovery [10]. To do that, they developed different configurations of the same simulation using different devices such as haptic devices, vision-based system and exoskeleton, and let a number of patients with chronic stroke randomly try these systems, with a treatment meant for be repeated in four weeks. While having overall good results in improvements with all of these three configurations, they noticed that less-constrained devices had a better result concerning the ability to retain achieved gains. In general, rehabilitation treatments

performed with Virtual Reality devices led to significant improvements for stroke support and recovery.

Figure 2.1: Virtual Reality for stroke rehabilitation. A Stroke of Genius: Neurorehabilitation through Virtual Reality; Illumin Magazine; 2019 Aug 27.

Given the heterogeneity of tests that have been done, some studies such as [11] and [12] have dealt with discerning between most of them to collect specific studies on this subject and find similarities between results and conclusions.

The first of these two reports dealt with collecting and summarizing the results of a set of experimental studies where researchers performed interventions using VR on patients with MCI or dementia, generally obtaining positive results. In summary, it was found that treatments with this technology have small-to-medium improvements on characteristics such us physical fitness, cognition and emotions; the results brought to the physical functions of some patients have indicated how VR treatments that also include physical training can benefit the motor functions of patients with MCI or AD. It is clear how the adaptability of this technology to the need of the patients, while trying out certain activities or tasks, results in a feeling of greater safety, comfort and less anxiety for them.

The second report devoted the greatest attention to the collection of studies that exclusively use Immersive Virtual Reality (iVR) for the diagnosis and treatment of patients with Alzheimer's Disease. In this analysis, it was clear that iVR can distinguish people with AD from healthy controls or people with other different

cases of dementia, while studies on treatments with iVR to improve the ability of performing certain routine functions are more limited. In general, however, patients with AD and MCI reported that they enjoy treatments executed with iVR, resulting in a positive factor for possible rehabilitation on participants with cognitive deficits. It was also reported that, for some tasks, learning and performance improvement of these patients growth more than other groups; among these, better performances were observed in space navigation tasks. In the report [13], the results of a 7-week treatment were observed for a patient with MCI and probable development of AD, who are asked to learn to orient themselves inside a symmetrical multi-storey building, without any landmarks or element of interests, using a control system composed by an Oculus Rift and a custom wheelchair, which captures real-world motion and reflect it in the virtual environment. Ah the end of the treatment period, the patient was able to complete every different orientation task in the simulation; beyond that, his family reported an overall improvement in his real-world orientation skills and navigation while driving.

2.2 Path finding

Artificial intelligence is playing a fundamental part in the development of modern video games, which bring with them ever-increasing requirements in terms of gaming experience and realism. The problem of pathfinding in video games is one of the most widespread challenges, as it requires developers to find the right compromise between the quality of results and the amount of resources required, considering the complexity of path search algorithms. Pathfinding usually refers to finding the best route from one point to another, taking into account obstacles and costs of traversing certain areas rather than others; the calculation of these paths can often take place at runtime, demanding it quite frequently. For this reason, there are numerous studies that aim to find the best solutions for these challenges, adapting to the environment they target, while optimising spent resources and speed of computation. While Dijkstra has long remained the only possible algorithm for finding paths within graphs, since the definition of the A* algorithm this has become the cornerstone for pathfinding, in games and non-games environment. Due to its versatility and the possibility of being refined to suit different tasks, there are multiple studies presenting variations of the algorithm.

In [14] the authors' aim was to define a pathfinding algorithm applied to robotics and video games, which would allow a path to be found that was short and realistic-looking. The proposed solution, the Theta* algorithm, is described as a compromise between the basic A* algorithm and pathfinding algorithms that make use of visibility graphs instead of grids. In the algorithm, each iteration is performed by making a line-of-sight check between the potential path nodes

and the nodes being expanded, continuing in the same direction by relying on the angles that define the range of visibility in the chosen orientation, imposed by the non-traversable grid cells, which represent obstacles.

The heuristic function that is used in A* to constrain the exploration of nodes to the most promising ones plays a fundamental role in the algorithm; for this reason, many of the variants that are proposed aim to intervene on this function in order to obtain desired results. In [15], the author tried to explore the possibilities of the heuristic function by introducing the concept of orientation into it, so as to take into account the direction taken up to that point for the choice of subsequent nodes visited. The results represent an optimisation in terms of time and visited nodes for the algorithm on uniform cost maps.

Figure 2.2: A grid map from a famous top-down strategy game. Eurogamer, 2017 Oct 19.

Although the concept of the uniform grid and algorithms based on it is a common and recurring problem in top-down strategy games and old RPGs, there are other representations of game areas over which paths are to be calculated. Extended game environments, such as open-worlds and MMORPGs, often rely on representations of the map by using a mesh that covers the traversable area, defined by a series of convex polygons, each of which ensures that a character can move freely from one point to another in that same polygon. Study [16] dealt with the implementation of an algorithm that optimises the complexity and resources used in defining a

minimum path over a mesh of convex polygons. The steps in this algorithm, after defining the area that can be traversed by the moving agents, are to find which polygons must be travelled through to realise the path, using the A* algorithm. Then, this channel formed by the detected polygons is used to calculate the shortest path within it, based on the concept of a visibility graph.

2.2.1 Maps representations for path finding

An element that connects all attempts to apply the path finding problem to game environments is the need for a usable representation of the map, generally composed of a structure of elements and connections between them. This kind of problem has been examined in depth in the past in robotics and motion planning applications [17], studying different methods of designing representative graphs of real environments where the movement of robots and automated vehicles had to be planned. Most of the methods proposed for defining representative graphs can be divided into two macro categories: cell decomposition methods and skeletonization methods.

The first of these two categories includes those methods that break down the environment under consideration into cells of fixed or variable size and generally polygonal or circle-shaped. Each cell often represents a navigable portion of the map, connected to the other cells to which it is adjacent. The most commonly used forms of cell decomposition are typically:

- Square cells
- Hexagonal cells
- Triangular cells

'Skeletonization' techniques deal with extracting traversable areas from the examined map topology by defining a set of vertices with their respective coordinates in the environment and connections joining those vertices that are in line of sight of each other.

These techniques generally produce representative maps composed of irregular cells that define particular graphs such as the visibility graph or the waypoints navigation graph.

2.2.2 Grids

A grid in a video game is generally a set of cells, each defined by a location on the map and representing a specific section of the environment, connected by edges to form a graph. The greater the granularity of the grid, the more accurate the

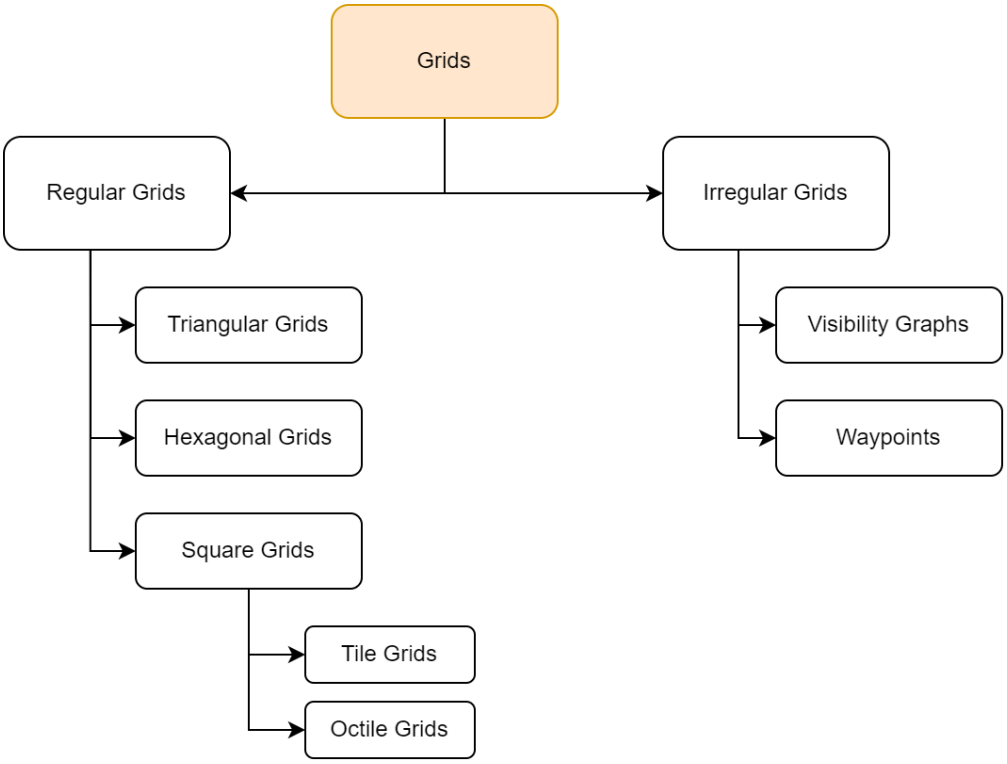


Figure 2.3: Most used map representation methods.

resulting description of the terrain. Following this, the fundamentals of two popular grid-based approaches are presented: regular grids and irregular grids [18].

Regular Grids

Regular grids are the best-known type of grids, used in video games and robotics, and consist of identical, equilateral polygons arranged to cover the entire map. Only triangular, square and hexagonal shapes can be used for regular tessellation of cells of equal size. The square cells are themselves classified into two sub-groups, depending on the number of connections found on each cell; if there are four connections, and thus possible movements from a cell, then the grid is generally called tile grid; if, on the other hand, movement is also permitted diagonally, reaching eight connections per cell, then it is called octile grid.

In study [19], a search is performed on all different grid types and results are evaluated by applying search algorithms such as A* and iterative deepening A* (IDA*). A grid called tex, consisting of square tiles but replicating the equivalent topology of a hexagonal grid, is also proposed. A number of considerations are made in the results, such as the fact that a hexagonal grid provides a better topological

representation of the underlying environment and how a tex grid brings with it the advantages of a hexagonal grid, but is simpler to implement. It was also shown how the choice of grid affects the asymptotic performance of the IDA* algorithm. In general, however, the requirements of the application and the design of the game influence which type of grid and search algorithm will achieve best results.

Figure 2.4: A square-grid tile map in a strategic videogame, Fire Emblem: Three Houses, 2019.

Figure 2.5: An hexagonal grid map in Civilization V, 2010.

Irregular Grids

Irregular grids have been increasingly used in modern times, providing the possibility of speeding up and optimising graph search algorithms at the cost of having to process a more complex map representation. The most applied irregular grid methods are the visibility graph and the waypoint system. Visibility graphs generally portray locations that are visible to each other, where each node in the graph represents a location point, and each edge represents a visibility connection between them. In practice, for every segment that links two points in the map and does not encounter any obstacles, a connection is generated in the graph. Often, to reduce the number of connections and thus the complexity of the resulting graph, vertices that would represent concave corners within the unobstructed areas are ignored. Similar to visibility graphs, and which make use of only convex polygons to represent individual sections of the game map, are navigation meshes, or Nav Mesh. Analogously, polygons included in the Nav Mesh represent walkable areas and adjacent polygons are connected to each other in the graph describing it.

The waypoint system, on the other hand, involves a set of points with their coordinates, the waypoints, and for each of them connections joining the same point to an arbitrary number of other nodes. The latter kind of system suffers from high computation times and wider memory usage. In [20], a waypoint system was used to plan routes for free floating space robots in a 3D environment. Dividing the task into two sub-problems dealing with finding the best sequence of waypoints to perform the required tasks and optimising the movement between them so as to make more efficient use of resources showed satisfactory results in improving the use of energy for such planning.

Figure 2.6: Waypoints representation (left) and NavMesh representation (right) in a World of Warcraft map section.

2.2.3 Gaming applications

In the context of video games, pathfinding problems occur repeatedly, receiving path calculation requests even several times per second. Taking games of the isometric MOBA genre as an example, the movement of the playing character takes place by clicking the destination on the map that one wants to reach, and this happens frequently over periods of seconds; each click corresponds to a new calculation of a shorter path to get to the destination, considering obstacles and areas that cannot be crossed. For this reason, the necessity for fast and efficient algorithms is further increased in this field.

Many studies have therefore been focused on the problem of pathfinding in video games, aiming to minimise computational time. In [21], a survey of pathfinding methods is conducted, going on to elaborate studies and modifications on the heuristic functions used in grid maps and applying extensions to the algorithms that allow the analysed map to be hierarchically subdivided and thus also decompose the problem into nested tasks. The latter analysis in particular often occurs in research of this type, proposing hierarchical abstractions of the planning problem on several levels, so that a more abstract solution can first be defined, which can then be refined into a lower-level solution later. These methods speed up algorithms by eliminating the possibility of backtracking on hierarchically higher levels.

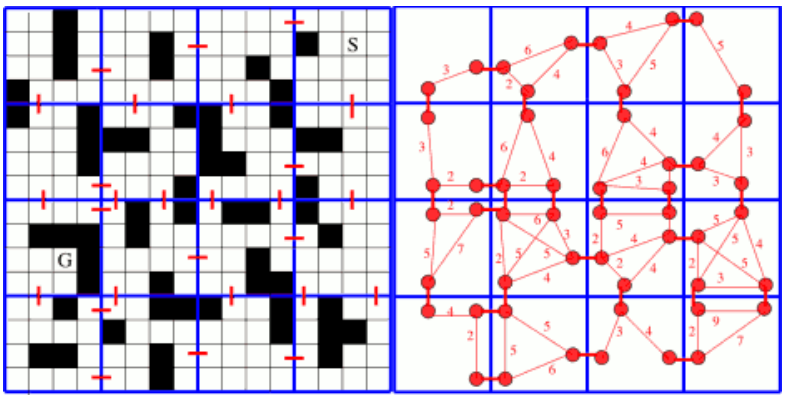


Figure 2.7: Hierarchical graph definition examples, Boost Hierarchical Pathfinding with Extended Graphs, Davide Aversa, 2015 Aug 28.

In regular grid representations, elaborations concerning the symmetry of certain portions of the grid can be made to speed up certain planning steps; with the assumption that the traversable zones on the map have uniform traversal costs, it is possible to reduce these zones to individual sub-paths with a cost equivalent to all those symmetrical paths that could be composed within them. Some reference is made to those solutions involving pre-computations of paths on the map that are stored in a database dedicated to storing, for example, all the shortest paths

joining certain nodes in pairs.

Furthermore, in this additional study [22], a combination of pre-computation criterion and hierarchical abstraction criterion for planning tasks was implemented to demonstrate the improvements these approaches can bring to the results. Starting with assumptions from the use of algorithms such as Partial-Refinement A* [23], one of the most famous algorithms that applies abstraction criteria, partial refinement and collaboration in pathfinding, combined with a pre-computed database system with information on map abstraction domains, they defined an algorithm called DBA*. The application of this algorithm to maps from a famous top-down tactical rpg produced noticeable results in terms of memory saved, in the range of hundreds of KB, and in terms of time saved, up to 10 seconds in some applications.

In practice, the DBA* algorithm performs an offline pre-computation before the actual path finding. In this phase, the algorithm processes the grid in such a way as to divide it into sectors of a predetermined size. Regions are then identified, which are sets of cells within a sector that are mutually reachable without leaving the sector. For each region, a representative point is defined and the database of paths joining each point of adjacent regions is pre-built using A*.

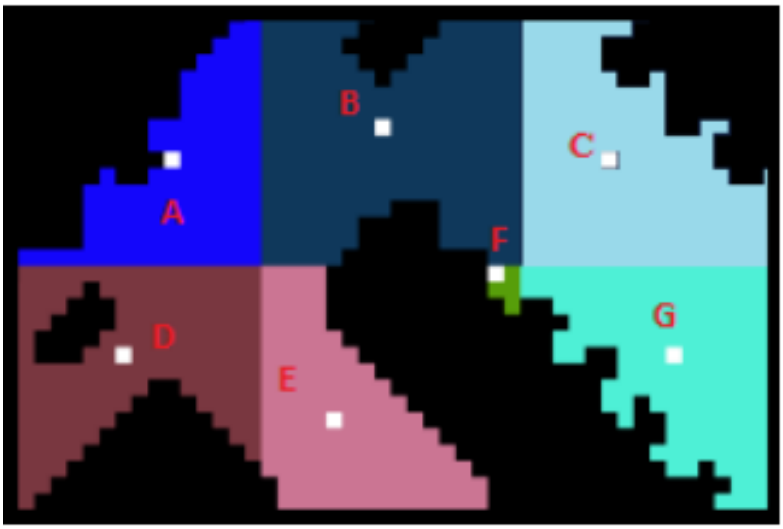


Figure 2.8: An example of the map abstraction with sectors and regions [22].

At this point, the online search uses the database to reduce computation time. The path found by the algorithm corresponds to the path that links the starting point to the representative point of the region that includes it, the path that links the representative point of the starting region to the representative point of the region that includes the destination point, and finally navigates from this representative point to the destination itself. A key feature of this approach is the possibility of using the time to reach the first representative point of departure as

the time to compute the route.

2.2.4 Objectives of path planning

In many applications, the objectives of path planning are not just limited to finding the shortest path between two points. Especially in robotics, but also in video games, the search for the best path aims not only to reduce the overall length of the result as much as possible, but also to keep away from obstacles, for reasons of safety or realism, and to define turns more softly in order to save robots more energy.

In [24], the authors addressed these multiple objectives to find the optimal path for the movement of a mobile robot. Defining the path as a sequence of segments, each defined by two points called rotation points that are part of the path itself, the resulting overall length equals to the sum of the lengths of the identified segments. The minimum distance between each segment of the path and each obstacle in the specific environment is then considered for safety purposes. Lastly, the average angle of the path is calculated, which defines the smoothness of that path, considering the angle of deviation formed by each pair of consecutive segments. The goal of the research was to define an algorithm that aims to minimise all these listed factors.

The authors demonstrated that the algorithm defined for this task, the Multi Objective Evolutionary Algorithm, achieves more accurate results than the well-known Particle Swarm Optimisation (PSO) algorithm. Its complexity, however, combined with the slowness of the operators involved in processing the factors to be minimised, make it suitable only for offline path planning.

Objectives of this kind are also often taken into account in video game contexts; in many categories, the problem of path planning is related to the presence of Non-Playable Characters (NPCs) populating a game environment. Pedestrians in a city, animals in a wild environment, or other vehicles in racing games are all subjects that must move through a specific environment with reference to paths. Their behaviour, however, must be realistic and natural, as well as optimising the distances travelled.

In this paper [25], the authors analysed the accuracy of algorithms such as A* and Dynamic Pathfinding Algorithm (DPA) with regard to the problem of finding the shortest path and avoiding dynamic obstacles during the movement of NPC cars in a car racing game. In the study, an attempt is made to combine the characteristics of these two algorithms to make the player's competing cars attempt to chase the shortest path while at the same time avoiding moving obstacles at runtime.

DPA is a method used to avoid obstacles along the route. It makes use of two collision points, located at the anterior part of the vehicle, so that they can detect

Figure 2.9: A famous car racing game.

obstacles located on the left or right front of the car.

Using Unity, a racing circuit was reproduced to test the effectiveness of the implemented method; the combination of the advantages of the two mentioned algorithms allowed the vehicles to always complete the circuit with static and dynamic obstacles. Starting with an offline path planning to be used as a guideline for the route the cars should follow, implemented with A*, the application of DPA at runtime allowed cars to avoid every obstacle they encountered.

Chapter 3

Technologies

3.1 Hardware

The objective of PEDAL simulations is to combine orientation and memory treatment with a physical training for the patient, using an integrated stationary bicycle. In the future, PEDAL software will run on a head-mounted display and a supported bike to read user inputs such as pedaling, steering, braking and interacting with any user interface, adding a PC external projection to let operators or relatives monitor session and progresses. To reproduce this system, the hardware team worked on a prototype of the stationary bike using easy-to-get components that allow them to simulate aimed functionalities.

Starting from a real bicycle, the team mounted a Smart Trainer on the rear wheel with a IR sensor module to capture its rotation and consequently the pedalling speed of the user. With this sensor, it was possible to use the read value as input for player movement in the software. The bike handlebar orientation was determined with a potentiometer sensor and used to change the direction of the player forward movement in the simulation, while the braking system was determined with a toggle switch. All these inputs values were managed by an Arduino Uno microcontroller, converted to have desired format and range, and then transmitted to the game engine.

At last, an HMD was used to immerse the user into the virtual environment. At present, there are many HMD proposals that combine excellent functionality with affordability; several state-of-the-art HMD systems were evaluated during the initial discussions regarding the overall hardware system in order to realise the first prototype for the project.

3.1.1 HTC Vive Pro

The first device that has been taken under consideration was the HTC Vive Pro. This head-mounted display was released in April 2018, and since then it has been one of the top of the range devices in the industry. It is a lightweight visor that is also sold individually without controllers, and requires direct connection to a PC. It features a high resolution, of 1440 x 1600 px per eye, an outward-facing camera and attachable headphones. The cost is slightly higher than other devices in the same range, and the need to have it connected to a computer all the time has led to an orientation towards other devices.

Figure 3.1: HTC Vive Pro headset.

3.1.2 HP Reverb G2

The HP Reverb G2 also appears as a device that needs to be wired to the computer in order to operate, but has an integrated tracking system that allows functioning without external cameras. In terms of resolution, it is the device with the highest configurations in this range, reaching 2160 x 2160 pixels per eye. It has front and side cameras that allow tracking of the user's movement. It provides developer support for Microsoft Windows Mixed Reality, with integration for all VR development systems.

Figure 3.2: HP Reverb G2 headset.

3.1.3 Oculus Quest 2

Oculus Quest 2, or more recently Meta Quest 2, is a stand-alone virtual reality headset developed by Facebook Reality Labs, formerly Oculus, in 2020. It has an Android-based internal operating system, and can run on desktop computer with an Oculus-compatible VR software, using an USB or Wi-Fi connection. This headset got a pair of goggle-like lenses that give the user a stereoscopic 3D visual and, paired with motion sensors and accelerometers, can reflect head motions real-time in the virtual world it is projecting.

Going on technical features, the Quest 2 runs on a Qualcomm Snapdragon XR2 chipset, with a display that allow a resolution of 1932 x 1920 pixel per eye.

Due to its portability, cost-effectiveness and the enormous developer support it provides, Meta Quest 2 was chosen as the visor for the prototype of the project.

Figure 3.3: Meta Quest 2 headset.

3.2 Software Development

When developing VR applications, developers should be able to focus on modeling advanced interactions and system behaviour, to make VR worlds become more realistic and responsive [26]. For that purpose, VR software systems, game engines and development kits are facilitating an abstract view of the system providing higher-level libraries and tools, reducing the effort needed to create and render virtual environments.

Modern game engines provide the developer with an editor for managing their assets, with a set of tools and APIs to optimize the development of game applications. The main component of a game engine is the rendering engine, which incorporates all of the complicated code needed to efficiently identify and render the player's view from a complex 3D model of the environment [27].

Beside the rendering engine, game engines include many other useful components to support the development:

- Physics and collision engine
- User interface support
- Animation engine
- Audio implementation engine
- Materials, textures, shadows management

- IDE to support programming with common languages

In VR development, the concept of toolkit describes a tool with the aim of providing reusable components that can be utilized to create VR application programs, avoid building everything from scratch, and reduce the amount of low-level programming [28].

On the other hand, software development kits (SDKs) are used by the developers to integrate VR applications for any specific platform. SDKs provide fundamental supports and are usually distributed by vendor and providers in order to help developers in the production of software applications aimed to run on their systems and devices. Typically, an SDK includes components such as:

- Libraries and APIs
- Processes
- Implementation and code samples
- Developer guides
- Blueprints

3.2.1 Unity

Unity is today the most popular game engine; with plenty of cross-platform features, it is popular with autonomous developers and triple-A studios. It includes professional tools for programmers and artists, it's easy to learn and proposes business models to favor small independent companies.

Unity uses a component-based approach revolving around objects called "prefabs", pre-configured game objects that can be stored in a project and utilized as templates to create new instances of the same object in an environment. While working with independent workspace environments called "scenes", it is possible to manage, move, edit, duplicate assets easily with drag-n-drop features, windowed game simulation, edit single or multiple instances of game objects working on components, attach script functionalities to the objects. Thanks to the massive number of user it has attracted since 2005, Unity now provides rich documentation and several videos and tutorials online. In more recent versions of the engine, tools for team collaboration has been greatly improved and integrated in the editor, with version control, cloud, branches, merging.

Another aspect of Unity that's distinguish this game engine from multiple others is the supported scripting language; leaning on the famous IDE Virtual Studio, it integrates C# scripts as components for its game objects. C# is an object-oriented programming language developed by Microsoft that combine basics from C++ and

functionalities from Java. Unity provides several C# libraries to help integration with all other components that participate to the lifespan of a game object in the scene.

The main components used in Unity are:

- Mesh Renderer, the component that allow the visualization of a mesh or a 3D model in the virtual environment.
- Rigidbody, it's a component used to implement physics to the object. Adding the rigidbody component to an object means this object will undergo gravity and collisions with other different objects.
- Collider, a bounding box used to manage collisions.
- Animations, a controller with machine-state system that let the object have different animations depending on user-defined variables.
- Scripts, one or multiple attached scripts that let the user define any functional behaviour for the game object depending on events, inputs, states.

Thanks to all different SDKs and a complex modern input system, with Unity is possible to manage all kind of user inputs, to integrate keyboards, mice, controllers, haptic devices, and microcontroller elaborated input such the ones coming from an Arduino board.

3.2.2 Custom tools for Unity

In Unity Editor, it is possible to implement customised tools to implement functions in the editor interface itself. In this way, there is the possibility of having additional supports in the UI that can be used both at development level and runtime.

In order to insert such a tool in the Unity UI, it is necessary to develop functions in a script that imports the UnityEditor library and place it in the 'Editor' folder in the project's root directory. Subsequently, a reference to the implemented tool can be found in Unity's 'Tools' menu, located in the top bar of the interface. This will open a customised window, which may contain user-defined variable inputs, a series of buttons to execute certain functions and other elements such as sliders, text fields and switches.

3.2.3 SDK for VR integration

SDKs for VR are important tools that allow easy implementation of pre-built and configured interactions in the project. The importance of interactions in a VR environment is emphasised, and SDKs generally provide supports for the integration

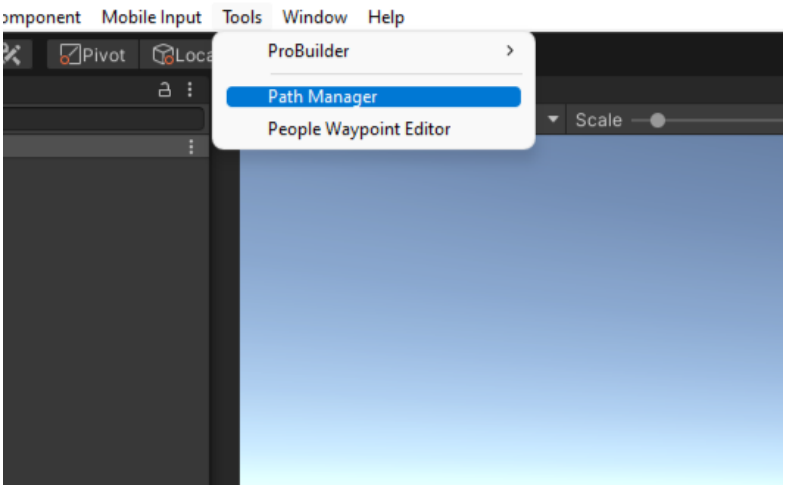


Figure 3.4: Unity 'Tools' dropdown menu with user-defined functions

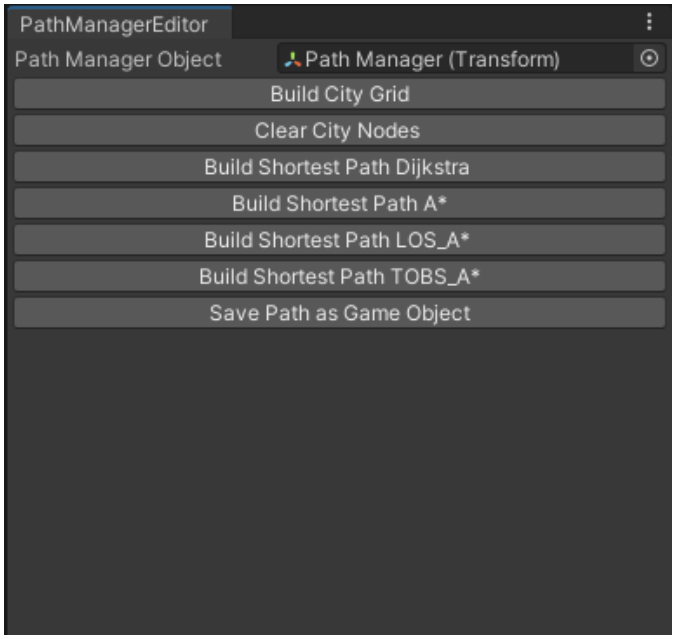
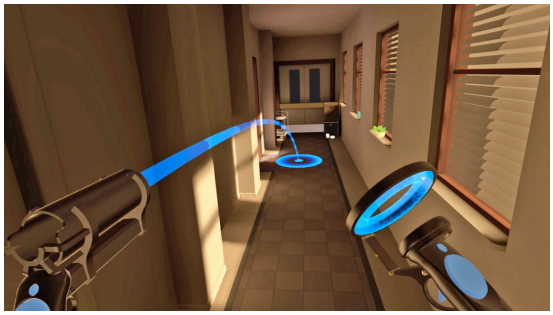


Figure 3.5: Path Manager Editor tool window, defined during development

of interactions such as grabbing objects, physically interacting and interacting with user interfaces. Other configurations that are provided by SDKs concern user locomotion; typically, player movement is implemented via one of the following methods:

- User teleportation to an aimed location

- Common controllers
- Hand tracking gestures



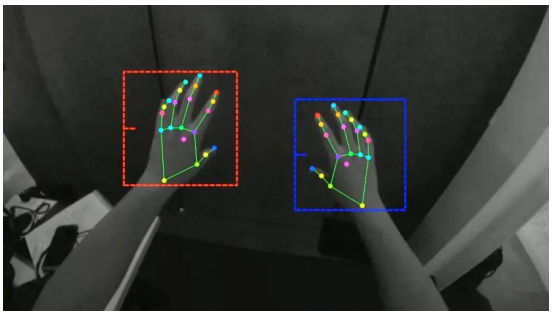


Figure 3.6: Movement control methods. Teleportation (left), Michael Eichenseer, medium.com, 2017 Jun 14, and hand tracking (right), David Heaney, uploadvr.com, 2022 May 02.

There are numerous SDKs developed for the most popular game engines; below a list of the most frequently used ones is presented:

- **XR Interaction Toolkit** is the official XR interaction framework made by Unity. It's a high-level, component-based, interaction system for VR and AR experiences. It allows 3D and UI interations to be handled by Unity input events, combining two types of components defined as 'Interactor' and 'Interactable' and an 'Interaction Manager' that is responsible for managing the two components together.
- **Microsoft's official Mixed Reality Toolkit** (MRTK) is a project proposed by Microsoft to provide components and functionality to aid the development of VR applications in Unity, initially released to support the Hololens system.
- **Virtual Reality Toolkit** is one of the first supports released for virtual reality. It provides configurations for locomotion, interaction with objects, physics of bodies in virtual space and more, while keeping itself low-coded and beginner-friendly.
- **Oculus Interaction SDK** is the official SDK release by Oculus. Its focus is oriented to the development on Oculus/Meta platforms, supporting components for standardized interactions for controllers and hands.
- **XR Tool Kit** (XRTK) is a community-supported branch of the Microsoft's MRTK, differing with the latter with its focus on an additional expandability and cross-platform support.

In this project, the XR Interaction Toolkit was added for the integration of the HMD Oculus Quest 2. In the development phases, relying on the Unity Input System, tests of the motion and exploration system were carried out using Oculus controllers. There are also plans for interactions via mouse and keyboard, in the case of parallel control by a treatment supervisor, and via an integrated input controller on the bicycle.

3.2.4 Unity input system

The Unity Input System is an extension package for Unity and represents a versatile tool for developing cross-platform input controls for any application. Through this system, it is possible to define standardised game actions to be called up in application scripts, without the necessity to discriminate the type of input, e.g. from controller or mobile. Taking care of this is the Input Actions system, in which it is possible to define the custom action and assign all those inputs from different platforms or methods that will correspond to that action.

All the actions defined by developers form the Input Action Asset, which represents the entire control scheme of the application. This is subdivided into a number of Action Maps, each corresponding to the control scheme that handles input in a certain context, such as in the game menu or during actual gameplay. In each Action Map, individual actions are defined, which may be of different types, such as buttons or boolean values, integers, two-dimensional axes, etc. For each action, the inputs that trigger this specific action are included; for example, by defining the action 'For ward', used in the application to move one's character forward, it is possible to insert input patterns such as the 'W' button of the keyboard, the up arrow, the analogue tilt of a controller pad. In the application, it will be sufficient to handle these types of input by referring to the event linked to 'Forward', regardless of the type of input behind the action.

3.2.5 Unity execution order

In Unity, each object in a scene has its own life cycle, which begins when the scene starts and ends when the object is destroyed. All the functionalities related to a game object can be distributed in the different event-related methods that constitute the cycle, related to initialisation, activation, updating in the current frame, etc.

Some of the most important methods that feature the execution of a script in Unity are listed below.

- **Awake** This functions is called as first, before any Start function is run, and just after a prefab is instantiated. The game object must be active or it will wait to execute this function until it is.

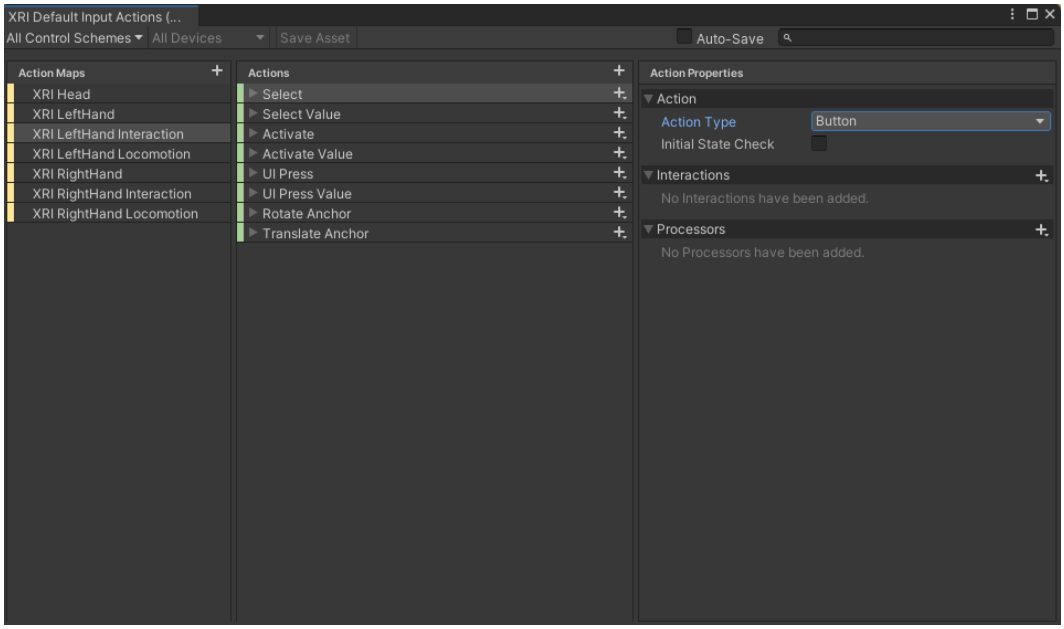


Figure 3.7: Input Action Asset of the XR player controller

- **Start** If the game object is active and the script is enabled, the Start function is called before the first frame update. Any object in a scene will run the Start function before any other Update function is called.
- **Fixed Update** This function is a recurring function that can be executed multiple times every frame, if the frame rate is low, or it may be not be called between certain frames if the rate is high. It is called on a certain timer, independent from the frame rate, and anticipate every physics calculation in the scene.
- **Update** This function is called once per frame. It is the main method for frame updates.
- **Late Update** This is called also once per frame, but after Update function has finished. Any calculation that are executed in the Update function will be completed when Late Update method will run.
- **On Destroy** This function is called after all frame updates, and for the last frame of the game object's existence in the scene.

Chapter 4

Development

4.1 Project requirements

Simulations for PEDAL rehabilitation have to be performed in a virtual environment that reproduces a real city area, developed in such a way as to be known, or at least familiar, to the patient. Working on the construction of every possible specific neighborhood would be an impossible task, and so, one of the rendering and graphics member of the project team worked on an algorithm capable of extracting, from a portion of a city sector selected from navigation map systems, a virtual reproduction of that area ready to be imported and used in the Unity editor.

The main idea is to select, from the imported area, a number of locations to be used as landmarks and to define starting points for different routes which the patients would be asked to travel, to simulate the task of returning back home from that specific location, such as from a cinema, a park, a famous attraction. The important requirement here is to have a logic of path definition that could be easily replicated, scalable, easy-to-use; it is needed a tool that allow developers to quickly define paths and start working on them soon. Although the real shortest route from a location to the destination is not always the correct one, due to realistic person behaviours such preferring wider roads with fewer turns, it is still a good starting point for path definition. By adding other criteria to recurring path search algorithm, it is possible to refine them in order to find routes that reflect people's realistic behaviour.

Analyzing the city section to decide which areas are viable and which represent obstacles should also be part of a simple task for developers, in a way that could be algorithmically replicated, fast enough to let them interact and modify the city as needed. The objective is to implement a Unity tool that allows, by entering a certain number of parameters related to the configuration of the city area, to construct the graph of nodes describing the parts that can be traversed and thus

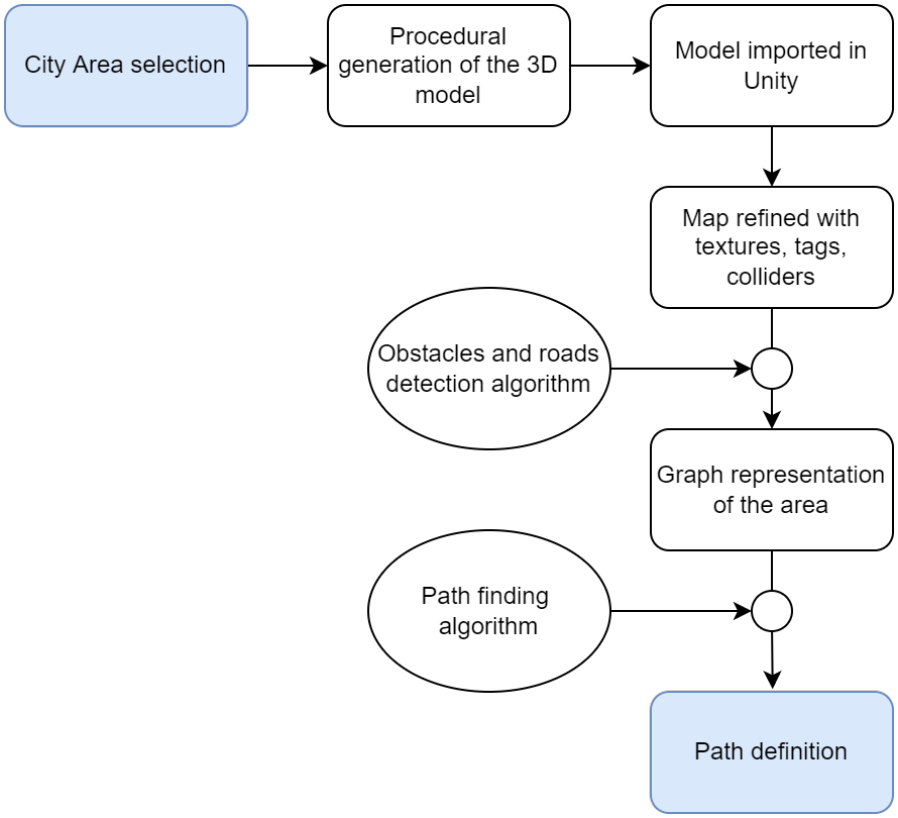


Figure 4.1: Path definition structure

subject to the path research.

PEDAL treatments are meant to be divided in different sessions; the first thing is to let the patient acquire familiarity with the system in its entirety, ensuring that they are in comfort and mentally prepared for the training. Since a virtual experience could be something new for the patient, it is expected to invite them into an exploring session of the virtual environment. Starting from the location that will represent the residence of the user, it have to be possible for the patient to have a look around, trying basic inputs of the system like steering and braking, while starting to memorize landmarks and particular characteristics of the environment around their virtual home. In this phase, called "Exploration", the patient could travel across the city environment for any duration, possibly discussing with a medical operator about bicycle configuration, headset settings, pedalling friction.

In a second phase, the patient will begin to learn a route home from the chosen location. At this point, a teaching system is required that maintains immersion in virtual reality and is clear and simple for users with potential cognitive difficulties. The patient will follow the directions given by a virtual assistant and/or a medical

practitioner and try to memorise the route so that they can then repeat it with fewer or no directions. The training sessions will not be time-limited, but should nevertheless be monitored in terms of timing and progress so that action can be taken in case of excessive mental strain on the patient, loss of orientation or motion sickness.

Thanks to the last phase, any improvements in the patient's treatment can be monitored. The objective of these sessions is to evaluate, by means of certain parameters such as elapsed time, the number of wrong turns and the total distance travelled, the user's performance in executing the chosen path. The idea is to plan for the patient, consistent with the type and intensity of his cognitive impairments, a treatment that would involve a fixed number of training sessions followed by a certain number of evaluation sessions, spread over several days. By studying the results and the progress in repeating the exercise, it will be possible to verify whether and how the treatment has positively impacted on the patient's state, so as to plan subsequent sessions by changing the route, the difficulty, and the type of help provided.

A further aim of the Pedal project is to allow a patient a repeatable exercise that can help both with motor activity and with preventing or reducing the degeneration of cognitive disease.

4.2 Tools developed for Unity

The first algorithm used in the path definition process that has been implemented is the Obstacles and Roads detection algorithm, to allow a fast processing of the virtual city and to identify those areas where the user will be able to move around by bicycle. For simplicity's sake, the idea is that the user will only be able to move on the road, thus excluding pavements and pedestrian areas. The other main algorithm is the one made to identify an actual path to be used in the training and evaluation phases, saving it in a Unity scene so that it can be manipulated by the other components of the virtual environment.

4.2.1 Obstacles and Roads detection

As defined above, a tool is needed that can analyse any reproduction of a city sector and schematise it into an abstract representation, composed by nodes identified as walkable or non-walkable. The first step is to find which parameters will define and differentiate each city area; working with rectangular sections, the maximum width and height of the map can work as boundaries for the scanning algorithm. The program for procedural generation of the virtual map takes care of extracting roads as separate objects, slightly elevated from the base plane, hence it is necessary to indicate the height at which the street objects are located.

At this point, it is necessary to define the requirements determining whether a node, which is located at a certain position on the map, is classified as walkable:

- **Absence of obstacles**. A node should be defined as non-walkable if an obstacle such as a building, tree or fence is present within its radius. The size of the area occupied by the node plays a key role in this function, as a radius that is too large would risk excluding entire parts of roads if located between buildings that are too close together or with decorative elements such as trees in unfortunate positions, while one that is too small could lead to the presence of small paths that are actually impractical.

- **Presence of the road**. A node should be defined as not traversable if there is no road within its radius. The user will have to move by simulating a bicycle course, so it should be recommended to only proceed on the street, aiming not to include areas represented by pavements, but also green parks, gaps between buildings, or 'grey zones' resulting from the city's procedural generation algorithm.

The competence area of a node is the area defined by the circle having a certain radius, called "tolerance radius", centred at the point of the node's coordinates. The dimension of the node, on the other hand, is the area covered by the cell it represents, in the grid covering the entire virtual surface, and this last parameter is responsible for defining how many nodes will cover the area of the city. Varying this parameter means increasing or decreasing the overall number of nodes that will describe the environment, and with it also the complexity of the operations that will then be performed on the graph. To easily represent the graph of nodes, these are saved in a two-dimensional matrix.

The total number of nodes defining a city section is, having h as the city height, w as the city width, where r is the length dimension of the cell represented by the node and n is the resulting number:

$$n = \frac{hw}{r^2} \tag{4.1}$$

The algorithm then takes care of scanning all the nodes created according to this criterion and checking their requirements to be classified as walkable. Any node that passes the two requirements and is classified as such is flagged as walkable and is linked to all its neighbouring nodes that are also walkable. A node is a neighbour of another node if the latter is adjacent to the former, also considering diagonal adjacency. Finally, it is created a graph of nodes and connections that can be navigated by means of row and column indices, so as to refer to the orientation of nodes with respect to their neighbours.

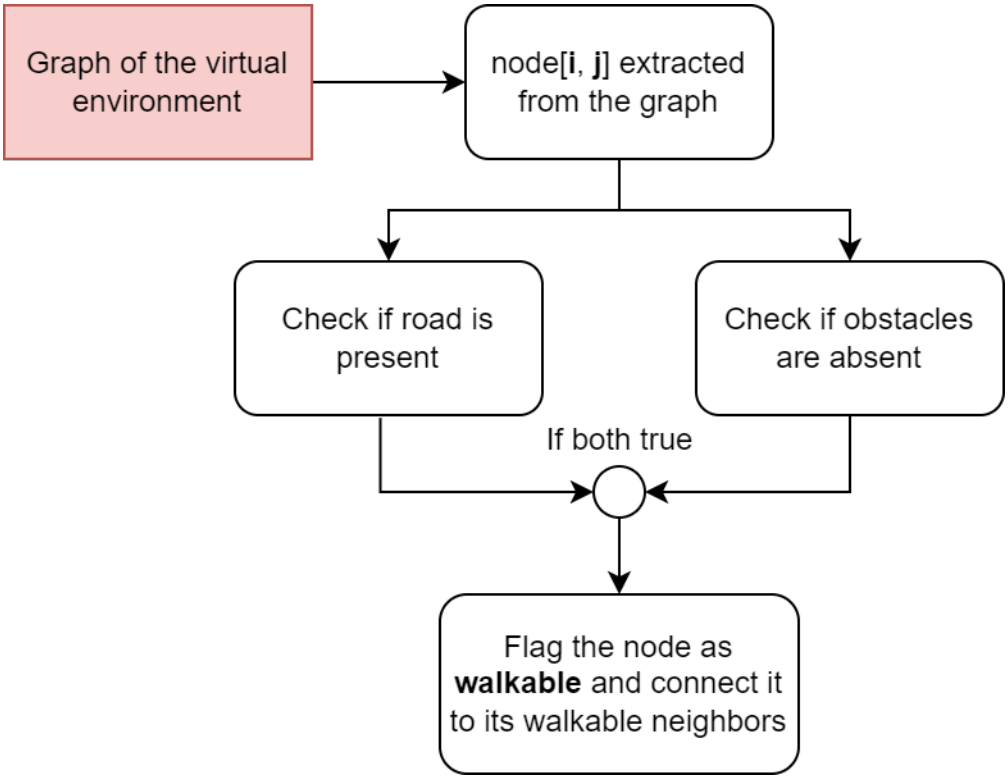


Figure 4.2: Single node step of the obstacles and roads detection algorithm

In Unity, it is possible to verify the previous requirements by making use of the Collider components, the object tag function and some dedicated functions of the UnityEngine library. As a first step, all roads intended to be included in paths must have a recognition tag, e.g. 'Road', which can be set in the Unity Editor inspector. At the same time, all elements constituting an obstacle must have an appropriate tag, which may be different between the various object types in order to facilitate their use with other functions, e.g. "Building", "Tree", "Fence". Then, for each node, the **OverlapSphere** function can be invoked, which creates a sphere, receiving as input the centre coordinates and the radius, and checks whether it triggers collisions with other objects in the scene. For each intersected object, it is possible to check its tag and thus ascertain whether it belongs to the above categories. The tolerance radius is used as the radius for the definition of this sphere; for simplicity, half the size of the grid cells has been used as tolerance radius in the development of this tool.

4.2.2 Path definition

To define a route that will be used in the training and testing phases, it is first necessary to identify suitable points to be used as starting points or destinations. Known locations or famous landmarks are appropriate solutions to represent starting points in order to simulate the task of having to return home after visiting that particular site. As for the destination, the first idea is to choose a location that is inserted in a living environment and use it as a fixed point, so that for the patient this will represent his home in the virtual world. However, it is possible to select a location as the destination with the same criteria as the starting points, reproducing the task of visiting two famous places within the city on the same day.

After deciding which starting and finishing points will constitute the itinerary, it is necessary to make some assessments of various factors; it is needed to check that the overall distance of the path, the number of turns, forks and crossroads, and the variability of the characteristics of the environment reflect the difficulty aimed at for this task. If these characteristics are suitable, the process can move on to the actual path identification.

4.2.3 Basics with Dijkstra algorithm

The first steps to verify the functioning of the path finding tool were made by applying Dijkstra's algorithm to the city graph. Dijkstra's algorithm solves the problem of finding the shortest path from a point in the graph, called source, to a destination. In the city graph, each node can have up to a maximum of 8 connections with neighbouring nodes, each of which will have a cost equal to the distance of the current node's co-ordinate point and the co-ordinate point of the neighbouring node it has connected. Specifically, each connection with neighbour nodes located on the same vertical or horizontal axis will have a cost equal to the size of each cell r, while it will have a cost of $\sqrt{2}r$ with the remaining neighbour nodes located in the directions of the diagonals. The sum of the costs of the connections between all nodes involved in the resulting path will represent the distance to be cycled to reach the destination.

Although functional, this algorithm is inefficient for tasks of this kind; in the case of graphs with a high level of symmetry and all equal costs, the shortest paths between a source and a destination are almost infinite. Dijkstra simply finds one of these, which in practice turns out to be variable in orientation and therefore unrealistic in comparison to a path a human would take to make the journey.

4.2.4 A* algorithm

A* is a search algorithm that has long been used in the pathfinding research community. Its efficiency, simplicity, and modularity are often highlighted as its

strengths compared to other tools. Due to its ubiquity and widespread usage, A* has become a common option for researchers attempting to solve pathfinding problems [29]. The special feature of the A* algorithm is the introduction of a heuristic function to determine a cost prediction for getting from a certain node to the destination. The potential of this function is due to the possibility of refining it and adapting it to the field in which the algorithm is to be applied, which is why it is very popular for the development of board games and strategy games.

Generally, when searching for a shorter path in a map, people tend to use as heuristic function the straight-line distance between the current node and the destination, the Euclidean distance, or in the case of grid maps they tend to prefer the Manhattan distance or the octile distance.

Algorithm 1: A* algorithm standard implementation

Result: Get path from start to destination
Initialize *node list* = [start];
while *node list* ! = [ ] **do**
  Select m from *node list* with lower cost;
  if m = *destination* **then**
    destination found;
  end
  Extract m from *node list*;
  for n *in neighbors*(m) **do**
    Compute the cost $f(m, n)$
    $f(m, n) = g(m, n) + h(n, destination)$;
    Where $f(m, n)$ is the cost of the neighbor n, $g(m, n)$ is the cost to move from m to n, and $h(n, destination)$ is the heuristic function, defined as the Euclidean distance estimated to be between the node n and the *destination*.
    if $g(m) + g(m, n) < g'(n)$ **then**
      Where $g(m)$ and $g(n)$ are the actual distance from start to respectively nodes m and n, while $g'(n)$ is the previous calculated distance to n from the starting point$n$.
      Update $g(n) = g(m, n) + g(m)$;
      Append node n to *node list*;
    else
      exit loop;
    end
  end
end

This algorithm, compared to Dijkstra, allows the destination to be reached faster by visiting fewer nodes. For the definition of the chosen heuristic function, however, the algorithm is focused on finding a path that, in the context of road maps, tends to follow the perimeters of the traveled area. In addition, the chosen route, of all possible routes, remains as similar as possible in terms of its course to the straight line connecting the source with the destination. This would mean, in terms of navigating a city environment, preferring roads with many turns and intersections to possible routes involving straight avenues and fewer turns.

The aim of the project is not to find exactly the shortest path, but to calculate a route and a distance travelled between two points that realistically reflects the journey that a person would take to make this trip. For this reason, the classical algorithm A* is not sufficient for our purposes, and it is necessary to refine its components, working on the heuristic function that defines the iteration of the algorithm.

4.2.5 Turns, Orientation and Bounds distance Sensitive A* algorithm

The study of paths within certain environments that have behaviours that can be traced back to realistic patterns, whether of people, vehicles or robots, is a much-debated subject that is constantly being researched and updated. In the case of [30], the aim was to generate paths for forklifts, working in warehouses, that were optimistically as smooth and with as few turns as possible.

Imagining a person cycling a route to a certain destination, it is easy to imagine how they prefer to avoid as many crossroads as possible, keeping to their side of the road, and making gentler turns.

It is necessary to redefine a new heuristic function h to account for these behaviours; each of the previous path-trending requirements is summarised by a function h_i, which is then weighted with w_i to be summed to the overall cost:

$$h(m, n, p, t_1, t_2, dest) =$$

$$[w_0 + w_1 h_1(m, n, t_1) + w_2 h_2(m, n, t_2) + w_3 h_3(m, n, p)]\, h_0(n, dest)$$

where m is the current extracted node of the iteration, n is the visited neighbor node, p is the previous extracted node and connected to m. t_1 and t_2 will be defined later with their respective functions. The weights corresponding to each contribution of the heuristic function are defined at the start, then varied by performing multiple attempts and analysing the results produced by their variations. The individual contributions h_i of the function are now described.

$$h_0(n, dest, a)$$

is the basic function making up the heuristic function; it is the Euclidean distance present between the neighbour node n and the destination node *dest*. To this contribution is then added a value a, which equals to the number of iterations of the algorithm that have been executed. In this way, it is possible to maintain approximately the same incidence of the contributions even for the nodes closer to the destination, which would see their Euclidean distance reduce considerably compared to the first ones.

$$h_1(m, n, t_1)$$

The function h_1 returns a binary variable that is equal to 1 if an obstacle is present within t_1 steps in the direction described by m towards n, and 0 otherwise. Those directions that would see a future obstacle on the path are then weighted more than the others, not taking into account the orientation that the user has previously maintained.

$$h_2(m, n, t_2)$$

The function h_2 returns a float variable defined between 0 and 1. This contribution takes into account the distance of the node n from the boundaries of the walkable area, and resulting value is closer to 1 the closer the node is to a boundary, considering up to t_2 nodes away. If there are then no boundaries within t_2 nodes from n, the value of the function returns 0, otherwise it returns $\frac{1}{d}$, where $d \leq t_2$ is the number of nodes between n and its nearest boundary. The aim of this factor is to prefer a path that stays away from the road edges, preventing the algorithm from squashing the path against the boundaries of the road.

$$h_3(m, n, p)$$

The function h_3 returns a float value defined between 0 and 1. This contribution is intended to ensure that the algorithm would prefer to maintain a stable orientation as long as possible by weighting changes in direction. The value returned by the function is equal to 0 if the direction taken from node m to node n is the same as the direction taken from the previous node p to the current node m; on the other hand, if this varies, the result of the function will be different from 0. More precisely, the function returns $\frac{1}{2}$ if the change of direction occurs only along one of the two axes, such as if the taken turn is only 45°, while in all other cases it will return 1.

The Turns, Orientation and Bounds distance Sensitive A* algorithm is then defined as described in the following pseudo-code:

Algorithm 2: TOBS_A* algorithm implementation

Result: Get path from start to destination
Initialize *node list* = [start];
Initialize $a = 0$;
while *node list* ! = [ ] **do**
  Select m from *node list* with lower cost;
  if m = *destination* **then**
    destination found;
  end
  Extract m from *node list*;
  for n *in neighbors*(m) **do**
    if *Obstacle in* t_1 *steps* **then**
      update $h_1(m, n, t_1)$;
    end
    if *Boundaries in* t_2 *steps* **then**
      update $h_2(m, n, t_2)$;
    end
    if *Heading change* **then**
      update $h_3(m, n, p)$;
    end
    Compute the cost $f(m, n)$
    $f(m, n) = g(m, n) + h(m, n, p, t_1, t_2, destination)$;
    Where $f(m, n)$ is the cost of the neighbor n, $g(m, n)$ is the cost to move from m to n, and $h(m, n, p, t_1, t_2, dest)$ is the updated heuristic function which estimates the cost to move from n to *destination*.
    if $g(m) + g(m, n) < g'(n)$ **then**
      Where $g(m)$ and $g(n)$ are the actual distance from start to respectively nodes m and n, while $g'(n)$ is the previous calculated distance to n from the starting point.
      Update $g(n) = g(m, n) + g(m)$;
      Append node n to *node list*;
    else
      exit loop;
    end
  end
end

4.2.6 Line Of Sight A* algorithm

Trying to think of having to summarise a route from a point to a destination, it is easy to imagine it as a series of checkpoints describing changes of direction, passing obstacles or choices made at a crossroads. For a navigator, the main interest in following a certain path is to know how one should behave when reaching these checkpoints, not considering the pattern that the nodes between the last checkpoint passed and the next to be reached maintain.

The objective is therefore to have an algorithm that is able to find path nodes that can represent checkpoints; to achieve this, it is sufficient to try to reduce the path description to nodes that can 'see' each other two by two. In practice, each node of the route must have its successor and predecessor in line of sight, no longer limited to its adjacent nodes.

To achieve this, the A* algorithm was modified to introduce a line-of-sight check between the neighbouring node visited and the predecessor of the current node. The implementation is shown as follows, excluding the sections of the A* iteration that remain identical in this version of the algorithm.

Algorithm 3: LOS_A* algorithm iteration

if *LineOfSight*(p, n) **then**
  if $g(p) + g(p, n) < g'(n)$ **then**
    Where p is the predecessor node of m in the path.
    Update $g(n) = g(p, n) + g(p)$;
    Update predecessor $parent(n) = p$;
    Append node n to *node list*;
  end
else
  if $g(m) + g(m, n) < g'(n)$ **then**
    Where $g(m)$ and $g(n)$ are the actual distance from start to respectively nodes m and n, while $g'(n)$ is the previous calculated distance to n from the starting point$n$.
    Update $g(n) = g(m, n) + g(m)$;
    Update predecessor $parent(n) = m$;
    Append node n to *node list*;
  end
end

The path obtained by using the LOS_A* algorithm, however, has certain characteristics that are not in accordance with the assumptions we have set up: although optimised, the path tends to pass very close to the obstacles it is intended

to circumvent. In addition to this, the distance obtained from the sum of the distances between the pairs of nodes is less than that obtained with A*, and consequently too optimistic with respect to a distance a user would travel to reproduce the route. Even though this approach is not suitable for defining a path for exercises and the evaluations that would be applied, the LOS_A* algorithm can be used for other purposes relating to the guidance and support system.

4.2.7 Path as game object

Once the path has been defined, this development-support tool can automatically insert it into the scene as a game object, saving its characteristics into it. The **CityPath** script will then be added as a component to the game object, inserting into it parameters such as the total distance of the path, all the nodes that constitute it, and their coordinates. Referring to the idea of summarising the path in a few checkpoints, it was possible to act on the path generated by TOBS_A* with the line-of-sight criterion; instead of introducing the LOS check within each iteration, the path is analysed after being extracted, checking for obstacles in the line between the nodes involved, taken in pairs. Only the furthest possible nodes that are at least in line-of-sight with the previous and the succeeding one, including the source and destination, are then preserved. The result of this operation is an array of key points that essentially describes the route between the starting point and the end point, where each of them is positioned in such a way as to identify a change of direction, acting as a guide for the person who wants to complete the trip.

The Path object is then described as follows:

- The parent object, defined by the **City Path** script. It stores the start and end point of the path, the total distance involved and the list of checkpoints that describe it, sorted from source to destination.

- A certain number of child objects, defined by the **Path Checkpoint** script component. In each of them, the corresponding previous node and the successor node within the path are saved, if they exist.

4.3 Progress and path tracking

During one of the phases of the PEDAL treatments, it is necessary to keep track of the user's progress with respect to the path by setting criteria for evaluation. The basic idea is to take into account what is the next checkpoint the user has to reach, and to base the indications and considerations on the player's progress in relation to it. The study on this aspect was carried out on parameters such as:

- The minimum distance before that checkpoint is considered reached. Since the important thing is that the user passes through the road within which the checkpoint is located, considering the extreme case in which the checkpoint is at the closest point to a boundary of the traveled area, e.g. the inside corner of a turn, while the user passes from the opposite point to the other boundary, the checkpoint should be counted as passed. For this logic, a value at least equal to the width of the road should be set as the minimum distance to surpass the checkpoint. In the case of an environment with roads of similar width, it is possible to set a fixed value for this parameter, while in the case of a more distinct difference between main roads, side roads or cycle paths, it is possible to consider setting a minimum threshold distance adapted to each checkpoint, depending on the width of the road where it is located.

- The maximum distance to the next checkpoint before considering the user on the wrong track. Should the user stray too far from the route, directions to the next checkpoint alone are no longer enough; instead, a path may need to be recalculated at runtime to help the user get back on the correct route. This parameter therefore defines when this type of event should occur, and at the same time when the user should be considered to be back on the correct path. Based on the same logic as the previous parameter, it is necessary to consider how the distance to a checkpoint may vary depending on the road the user is on. For example, after taking a very long avenue, the user could find himself at a great distance from the next checkpoint, although remaining on the correct path. Again, designing this parameter for each checkpoint based, this time, on the length of the road on which it is located is the right solution. For example, the maximum distance should not be less than the distance from the previous checkpoint to the target checkpoint. In general, this parameter alone is not sufficient to evaluate this criterion, and it is therefore necessary to add other evaluation parameters, such as orientation, last cells crossed, etc.

The first parameter was used to define the threshold for which the update of the checkpoint defined as 'next to be reached' takes place. Specifically, when the user enters the range defined by this parameter, a function updates the status of the checkpoint reached, marking it as passed, and picks its successor checkpoint, which is used as the next target by the navigation functions. Referring instead to the maximum distance parameter from a target checkpoint, the virtual assistant intervention was implemented to ensure that the user is guided back to the correct path. To guide him, an algorithm for calculating the shortest path is applied, using the user's position as a starting point and, as destination, a checkpoint selected from the last of those visited or the next checkpoint to be reached; this selection depends on whether the purpose is to make the user repeat part of the path, so that they travel it in its entirety as it is designed, or whether it is sufficient to get

them back on the right path, accepting a diversion on the route.

4.3.1 Virtual Assistant behaviour

A virtual assistant (VA) was included in the project in order to implement certain functionalities; the first of these was to integrate path tracking with the VA, so that directions could be given to the user through it, while the second was to give the possibility of simple interactions with the system, aiming at minimising the patient's effort (button presses, menu complexity, etc.). Focusing on the function of guidance and directions, the first step was to define the criteria by which directions are provided to reach a specific checkpoint, depending on the current state of the user. Two types of information on the user's state in the environment were identified:

- The most recently covered road, called "Headed Direction". This information is described by a 2D vector showing part of the road the user has travelled, retrieved by averaging the vectors obtained joining the coordinates of a number of nodes visited by the user and the user's position. The number of nodes to be considered for the calculation is a parameter that is defined according to the characteristics of the environment and the road.

- The user's current "Orientation". Regardless of the road taken, this information is defined by the component in the horizontal plane of the user's 'forward' vector. Although this orientation is not relevant in the description of the route taken so far by the patient, as they may have, for example, turned on the position to look around, it is important with regard to the directions to be given. In practice, providing the user with the simple indication that they should "turn right" may not be sufficient, or even misleading, if the user is oriented very differently from the direction of the path taken so far.

Thanks to the work of rendering and animation team members, certain types of indications have been implemented which the virtual assistant, realised as a small floating robot, can provide to the user in real time. With regard to orientation, animations were created to indicate to the right or left with respect to where one is facing; in general, this type of indication is used to suggest to the patient which way to face in order to return in the right direction on the path. This type of VA behaviour is performed by analysing the angle between the vector indicating user's orientation and the vector connecting user's position to the position of the target checkpoint; either the actual checkpoint of the path to be reached or a 'support' checkpoint generated by the runtime pathfinding algorithm aiming to return to the main path.

The other type of indication that has been realised makes use of graphic effects integrated with the VA that indicate whether the road being taken is correct. If this is not the case, such as if the user is too far from the target waypoint or his heading direction differs too much from the correct direction, signals suggesting the wrong course are displayed.

Combinations of these indications, supplemented with other supports such as textual explanations, navigation maps, notification sounds can be used in the various treatment phases for the intended purposes.

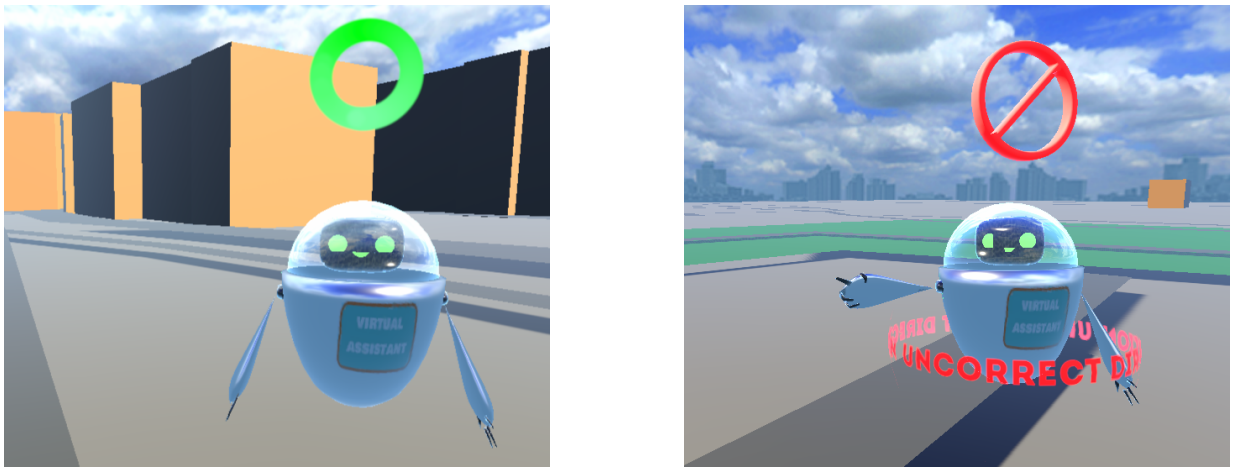


Figure 4.3: Idle animation with correct graphical effects (left) and pointing left animation with incorrect graphical effects (right)

4.3.2 Evalutation criteria

As mentioned above, during sessions, it will be necessary to be able to evaluate the performance of a user attempting to reproduce the defined path. While many criteria will be based on behavioural parameters of the examinee, others will concern movement parameters such as the total distance travelled and the time taken to reach the destination; these two simple criteria form the basis of the characterisation of a path. The objective of defining a realistic path also helps to produce an equally realistic estimate of the distance corresponding to the given path, so as to have an effectively replicable value that corresponds to a distance that, on average, a user who is perfectly familiar with the itinerary would cover. By referring to this distance, it is also possible to estimate the time corresponding to this optimal case by taking into account the average speed of the user's pace in the virtual environment.

Determining the speed of the user's motion is not trivial, and will require special attention. In an advanced state of the project, it is possible to imagine a movement

speed that depends on the patient's pedalling speed, setting a upper limit for this value. At the same time, it is not intended to set a speed system as accurate as that of a driving simulator, as the objectives of these treatments are different. To begin with, a static speed parameter was defined by evaluating a few factors. What is needed is a travelling speed that allows users to easily take the tightest curves in the game map and also allows sufficient time to orient themselves by looking around without stopping. At the same time, it is necessary that the speed of movement is sufficient not to make it too tedious to pass a long avenue without turning.

In Unity, it is generally recommended to refer to a unit of measure for the length of the virtual scene as a metre. In this way, in addition to reporting the elements of the environment with proportionate and real dimensions, it is possible to define the speed of the agents navigating the map in terms of metres per second. After several attempts of different velocities, it was decided to use $\mathbf{v} = 4m/s$ as the speed parameter for further experiments.

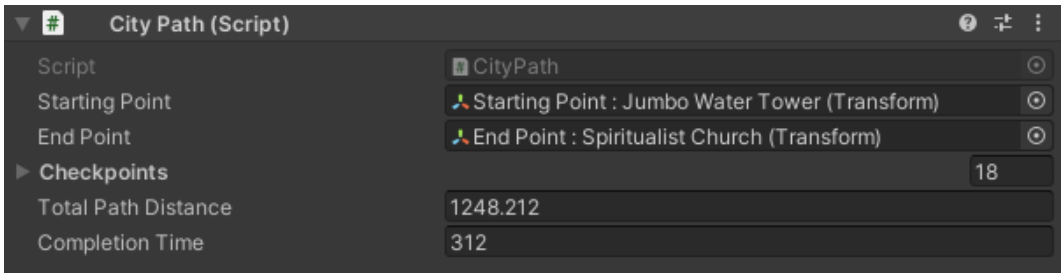


Figure 4.4: A path script with distance and average time displayed.

In the following chapter of this document, relating to tests and results, data taken from a number of tests are reported, aimed at verifying how similar a path defined by using methods described above is to an optimal path that a user would take being conscious of all the directions that constitute it. The data collected refer to the total distance travelled by the user during the journey and the time it took them to complete it.

4.4 Pedestrians system

The realism of the virtual environment is an important component to increase user immersion. Where the accuracy of the 3D models, the reproduction of the player's movements, and the interaction with the environment play an important part in the success of this objective, the introduction of non-player characters in the virtual world aims at making the user perceive a more alive and somewhat less simulated place. The objective in this case is to develop a system to manage

NPCs in the game environment. The first consideration to be made is how many NPCs to place in the map, aiming to find the right compromise to have a realistic number of people 'living' the environment, but without overburdening the game engine with a high number of models. A second consideration relates to how it is sufficient to place characters only in the player's immediate surroundings, where they can therefore be seen, leaving distant areas empty. With these observations, it is possible to define parameters to algorithmically manage spawning of pedestrians in the virtual environment:

- The number of pedestrians to be brought into the explorable area.
- The maximum distance range within which pedestrians should appear.
- The maximum distance within which pedestrians can remain in the environment, beyond which they would then be removed in favour of the appearance of new NPCs.

4.4.1 Pedestrians spawn management

Using the parameters defined above as input for the function, a script was defined for managing the appearance of NPCs in the game map. The question that has been raised is where to make the characters appear and where they could move; as first, all those areas where there are no obstacles, such as buildings or trees, but where there is no road, designed to be dedicated to vehicles, have been chosen as areas that the NPCs can travel through. Therefore, the Obstacles and Roads detection algorithm was re-proposed, but with a variation: the nodes marked as walkable for NPCs are those nodes that aren't walkable by the user, they do not have any obstacles and among their adjacent nodes there is at least one node that is walkable by the user. In this way, all areas representing pavements adjacent to the road travelled by the player are preferred as spawn points.

In Unity, there is a component that autonomously manages the path an NPC will take when setting a certain destination, called the 'Nav Mesh Agent'. Thanks to it, it is possible to enter parameters such as speed, acceleration, and angular velocity that the object in question must have in the scene; it is then only necessary to insert the destination that the NPC must reach in order to allow it to move through the environment. To choose the destination, a function was defined that randomly picks the position of a node that meets the 'no obstacles and no road' requirement.

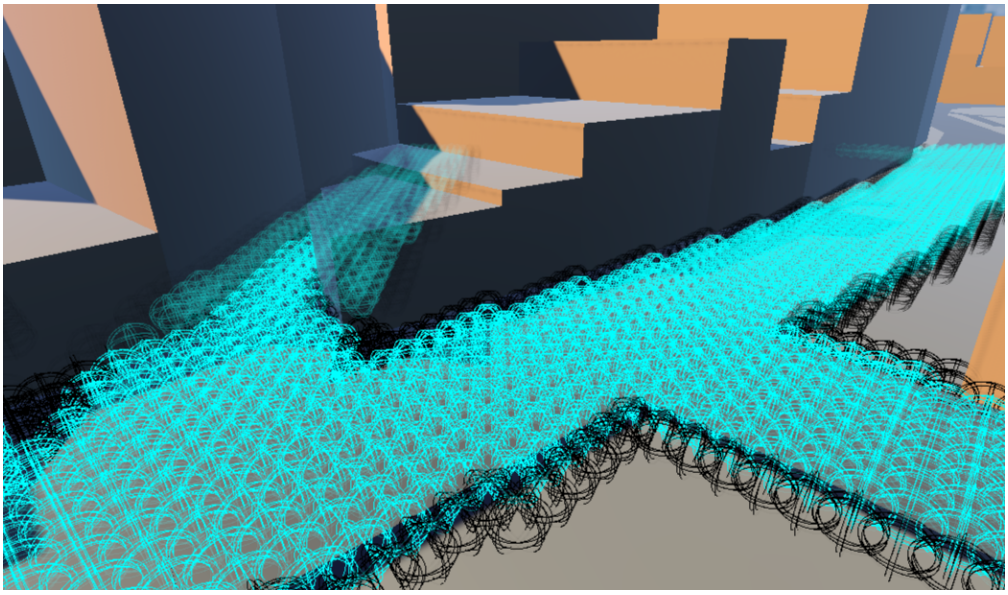


Figure 4.5: User walkable nodes (Blue) and pedestrians spawning nodes (Black)

To manage the actual NPC spawn and vanish functionality, two functions were defined that rely on Unity's 'Coroutines' system. A coroutine makes it possible to divide several tasks between different frames, returning control to Unity at the end of each frame, and restarting in the next frame from where execution had been left off.

Defined functions, managed by two separate coroutines, are called 'Spawn' and 'Despawn' and have been implemented as follows. Variables that are defined are p as the player current position, $maxCount$ as the maximum number of pedestrians that can be alive in the environment, $activePedestrians$ are the list of all current NPCs that are located in the game world, the range r_1 is the maximum distance from the user where pedestrians can appear, while r_2 is the maximum distance from the user current position where pedestrians can remain in the environment.

Algorithm 4: Spawn function

```
while true do
    if sizeof(activePedestrians) < maxCount then
        Select p_r as random position taken from one of the nodes identified
         as possible spawn points for NPCs and at a maximum distance of r_1
         from the user's position;
        if p_r is found then
            Pick a random NPC model m from the available model list;
            Instantiate npc with model m and position p_r;
            Add npc to activePedestrians;
        end
        Wait for end of the frame;
    end
end
```

Algorithm 5: Despawn function

```
while true do
    for npc in activePedestrians do
        d = distance of npc from p;
        if d > r_2 then
            Remove npc from activePedestrians;
            Remove npc from the scene;
        end
    end
    Wait for 1 seconds;
end
```

4.4.2 NPC behaviour

For NPCs, several possible behaviours that they may have in the scene during their presence on the map were defined. After collecting some animations to represent these behaviours, the criteria for choosing which of these to perform, with what possibility and for how long were thought of. The possible behaviours, called 'statuses' of NPCs, were divided into two main categories:

- The 'main' statuses, i.e. the behaviours that can occur starting from an idle state of the NPC, such as at the moment they are spawned in the scene.
- The 'secondary' statuses, are the behaviours that can occur as a result of specific events that happen during one of the main statuses or other secondary statuses, thus interrupting their execution in favour of the new behaviour.

The functioning flow of the behaviours was implemented starting from an idle state for each character; when in this state, one of the defined main statuses is randomly chosen, the variables necessary to realise it are set and the animation is then started. At this point, the chosen behaviour could be completed without interruption, e.g. finishing the set time or reaching the chosen destination. However, one or more events may occur, and each of them has a previously defined possibility of interrupting the current execution and switching to a secondary status. The values describing the possibilities of each status being chosen are parameters that have been tuned through several simulation attempts. When one of these statuses, whether main or secondary, is completed without interruption, the NPC returns to the idle state and the cycle begins again.

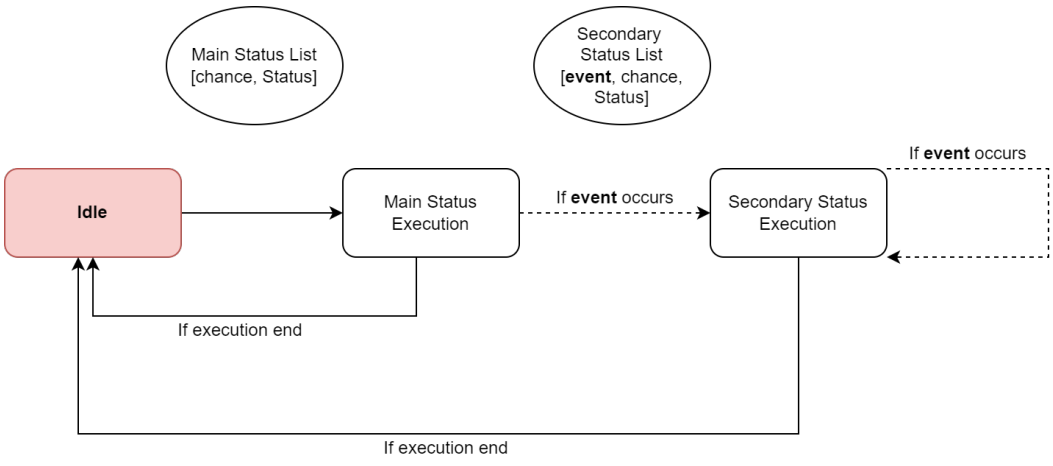


Figure 4.6: NPC behaviours diagram

For first simulations, a total of four possible statuses were implemented, divided into two main and two secondary ones. Specifically, the two main behaviours

an NPC can have are walking and calling on the phone. The first of the two continues its execution until the destination is reached or until an obstacle prevents it from proceeding; the second status has a fixed execution time of 60 seconds. The probability values of primary statuses are indicated by a corresponding value p_i, which when all are summed up results in $\sum p_i = 1$.

As for secondary behaviours, chatting with other NPCs and observing buildings and other landmarks on the map have been developed. Both of these statuses can be undertaken by characters as a result of events occurring during the walk status, with a possibility indicated by a certain value p_j, recalculated at each occurrence of their respective event; this value is independent for each status and is $p_j \leq 1$.

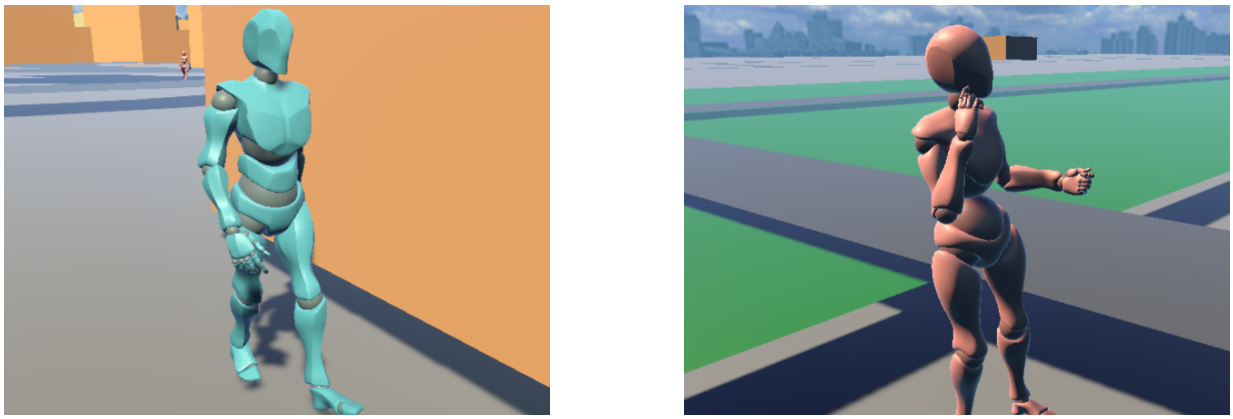


Figure 4.7: Walking status of an NPC (left) and Talking status of an NPC (right)

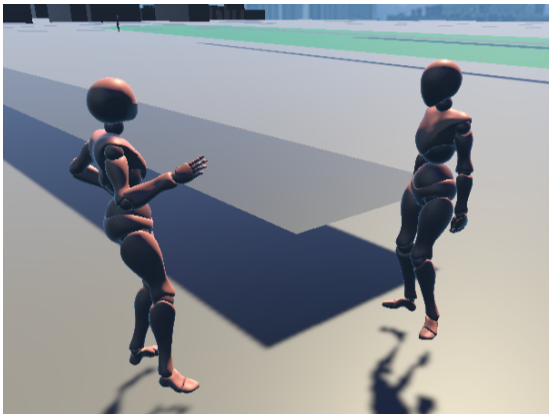


Figure 4.8: Chatting status of two NPCs

Chapter 5

Test and Results

5.1 Path definition on a virtual environment

The application of the TOBS_A* algorithm led in most cases to the definition of paths that reflected the requirements. In order to present some results, a reduced virtual environment with streets, pavements and buildings was realised to emphasise the differences the applied pathfinding algorithms demonstrate. The source and destination of the path used are the same for all applications, aiming to expose how variations in defined parameters affect the result.

For each attempt at path definition, the characteristics that characterise that attempt will be shown, with screenshots of the nodes describing the environment grid. Each node is shown in light blue if it is walkable by the user, in dark blue if it is included in the path detected by the algorithm. Nodes belonging to non-walkable areas will be left hidden. Figure 5.2 shows the result of the application of the Obstacle and Roads detection algorithm, which identified the nodes that could be visited by the user during simulations. It is worth noting that the entire road dedicated to vehicles was covered, leaving out the areas corresponding to pavements.

Figure 5.1: Virtual environment representing a small neighborhood built with Unity

Figure 5.2: Graph representation of the walkable areas (light blue) of the map

5.1.1 Dijkstra algorithm application

As a first example, Dijkstra's algorithm was applied to find the shortest path from the starting point to the finishing point, located in the southernmost part of the game area. By representing in green all those nodes that have been visited during the iterations of the algorithm, it is easy to see that almost the entire playable area has been visited, before finding the actual destination. This does not prove particularly efficient, especially in the case of very large maps with a high level of detail (i.e. very smaller nodes).

Figure 5.3: Path defined with Dijkstra algorithm

As is evident, Dijkstra's algorithm per se is not a good alternative for the pathfinding problem applied to graphs with homogeneous link costs and symmetrical connections between all nodes.

5.1.2 A* algorithm application

Following the logic made during development, subsequent attempts at path finding were conducted with the A* algorithm. Thanks to the heuristic function applied, in this case the Euclidean distance, as the nodes are very small compared to the map size, it was possible to find the destination faster. The overall nodes visited are significantly smaller than those visited with Dijkstra, and are shown in figure 5.4. The resulting path, however, is identical to the previous case.

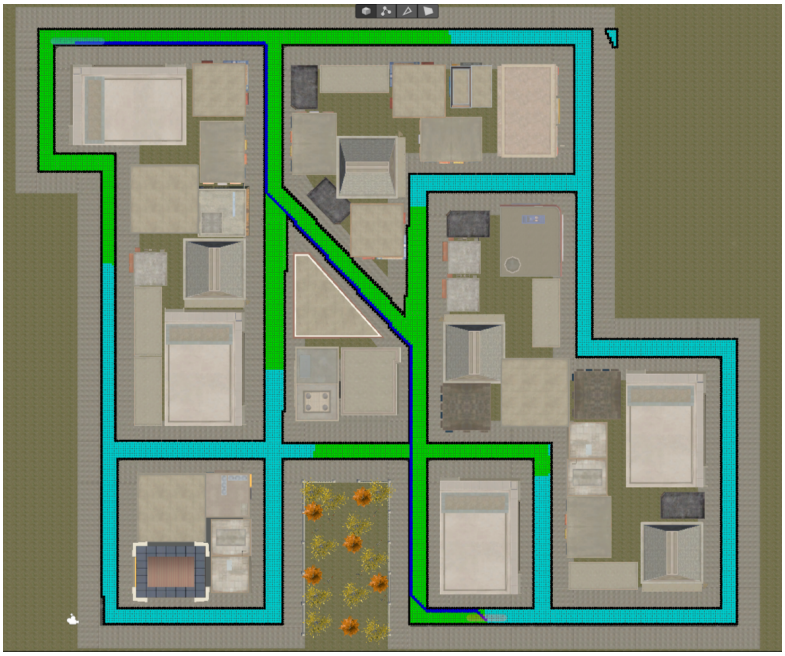


Figure 5.4: Path defined with A* algorithm

As expected, however, the path resulting from this algorithm tends to remain adjacent to the boundaries of the walkable area, deviating from the pattern that would be expected by a person making that journey by bicycle.

5.1.3 TOBS_A* algorithm application

Arriving at the application of the TOBS_A* algorithm to the reproduced virtual environment, it is possible to bring to light the main influences that the weight and support parameters of the heuristic function have on the realisation of the final path. The parameters that were manipulated for the experiments are listed again below:

- The weight of the basic distance component of the heuristic function w_0
- The weight w_1 of the function component h_1 that takes considerations of the presence of obstacles in the direction of the node n, with the number of steps performed in that direction equals to t_1
- The weight w_2 of the function component h_2, which impacts on the function that checks if there are any boundaries of the walkable area near the node n, checking in a range of nodes from n equals to t_2
- The weight w_3 of the function component h_3, which affects the total cost from m to n whether the movement between these two nodes would result in a change of orientation from the one previously held

The value of w_0 was kept equal to 1 in the experiments conducted, to emphasise the impact of the other components on the resulting cost. Next, the results of a number of experiments performed by varying the w_i parameters are presented and some considerations that led to selecting the final value of these parameters are explained.

For all tests, values for parameters t_1 and t_2 were set according to the conformation of the road; taking into account the width in nodes of the roadway, t_1 was set equal to 7, while t_2, representing the desired distance from the sides of the road, equal to 2.

In the first test, shown in Figure 5.5, the values of the weights $w_1, w_2, w_3 = 1$ were set. Although the resulting path appears acceptable, some considerations can be made. In some sections of the path, it was kept adjacent to the boundaries of the road, in order to favour an orientation without future obstacles. It is therefore clear that, according to the desired criteria, the incidence of the h_1 component must be less than the other components, aiming first to have a path that tends to stay more towards the middle of the road.

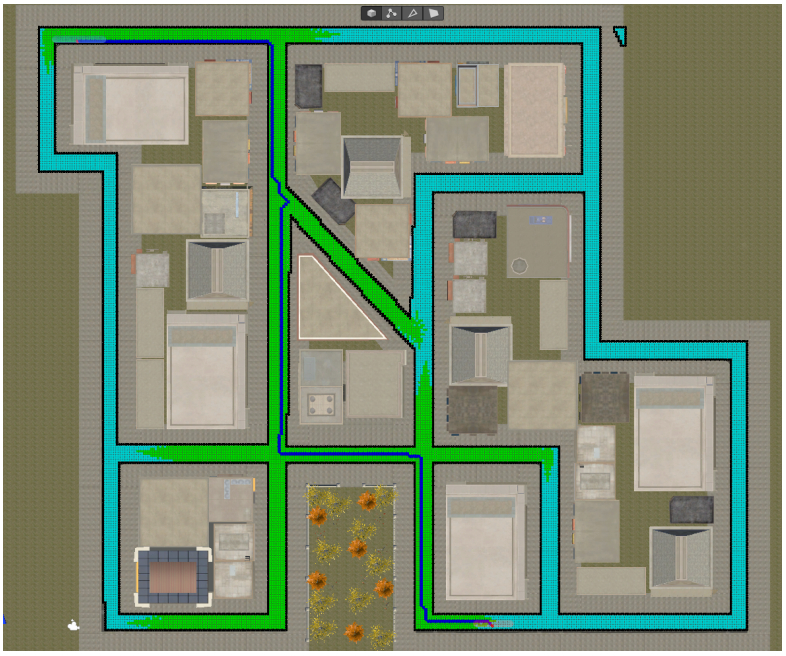


Figure 5.5: Path defined with $w_1 = 1, w_2 = 1, w_3 = 1$

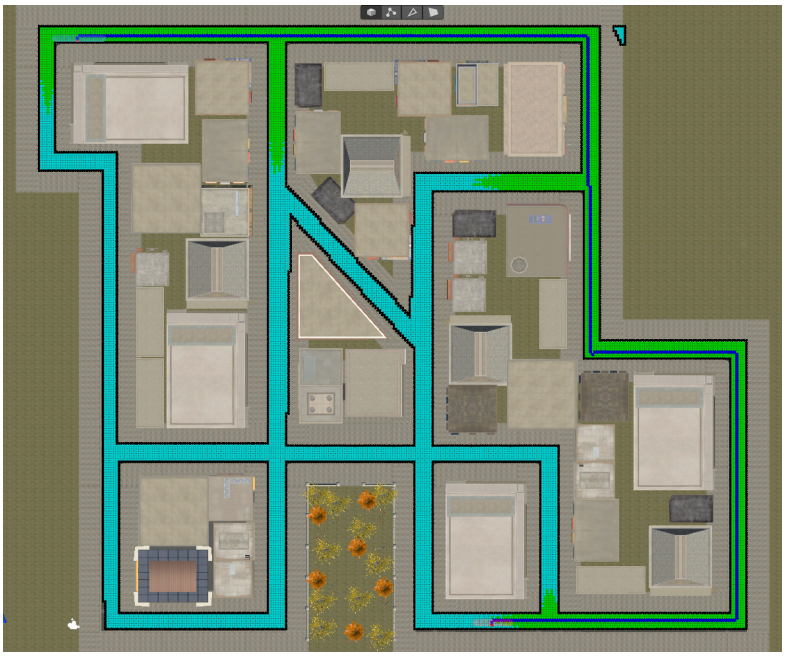


Figure 5.6: Path defined with $w_1 = 0.3, w_2 = 1, w_3 = 1$

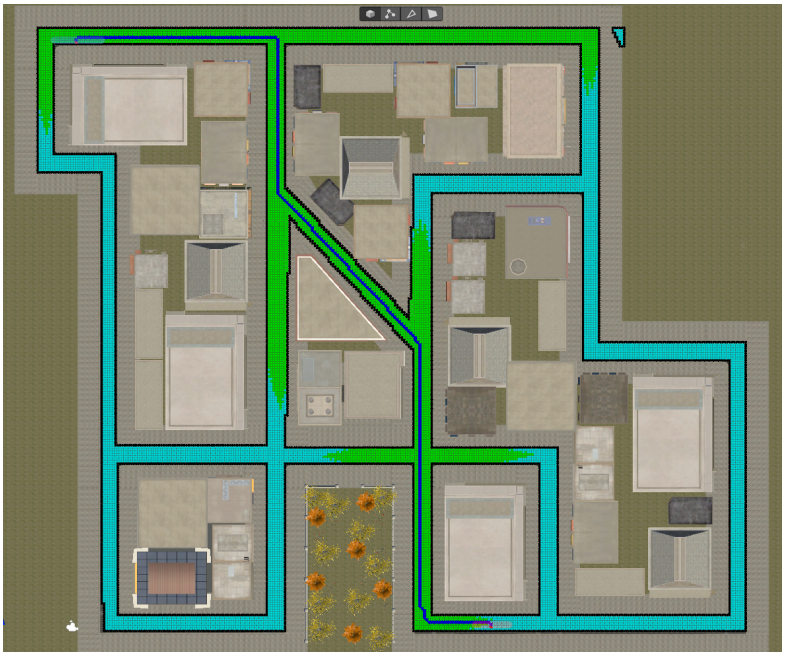


Figure 5.7: Path defined with $w_1 = 0.3, w_2 = 1, w_3 = 0.5$

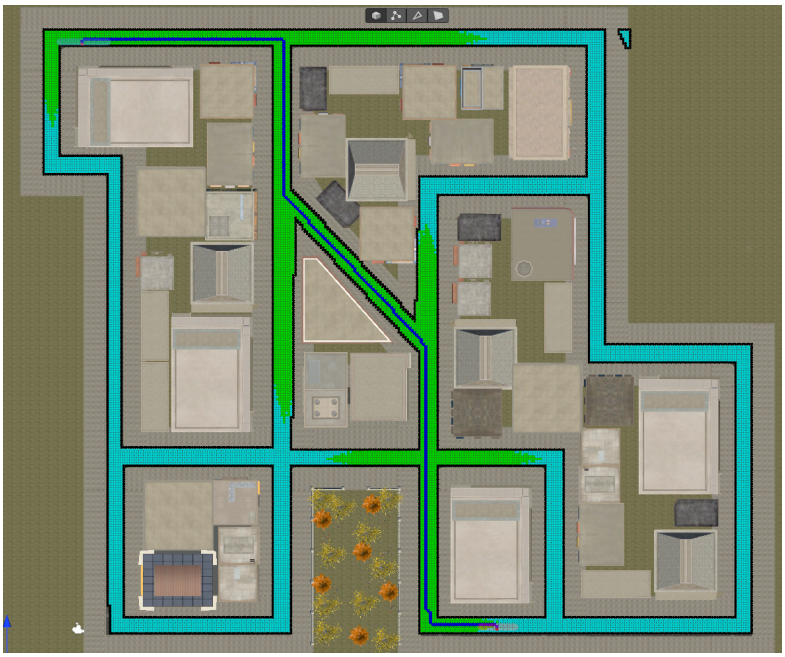


Figure 5.8: Path defined with $w_1 = 0.3, w_2 = 1, w_3 = 0.1$

For a second test, therefore, the weight w_1 was reduced to the value of 0.3, leaving the other two unchanged. In this case, the criteria of distancing the path from the boundaries was fulfilled, however, showing another problem. The defined path does not follow the course of the shortest path from the starting point to the destination, but favours keeping the same direction as much as possible, lengthening the route. In a city layout such as the one presented, where the streets are all roughly similar, the route in figure 5.6 is unrealistic in this respect. According to this result, the effect of the h_3 component should also be lowered, especially compared to the incidence of the base component h_0.

Lastly, some tests were conducted by varying the parameters w_1 and w_3, keeping them lesser or equal than 0.5, to analyse the relevant results. Figures 5.7 and 5.8 present two examples of paths realised with these variations, respectively $w_1 = 0.3$ and $w_3 = 0.5$ in the former one, and with $w_1 = 0.3$ and $w_3 = 0.1$ in the latter one. The path found is more or less identical in both tests, fulfilling the established criteria and thus producing a path with a realistic pattern and appropriate length to be implemented in the project. A small variation occurs in the number of nodes visited before arriving at the solution, although this is of little significance.

5.1.4 Paths definitions on realistic environment

After making these conclusions, the path definition system has been brought to a map with a more realistic conformation: a section of the city of Colchester. Colchester is a town located in East of London, England, and a rectangular section of it was reproduced as a virtual environment for the simulations. It has been made sure to include some famous locations within this section, such as Colchester Castle. The idea is to perform parameter variation tests in such a context, aiming to find the most suitable values for multiple path definition in this environment. To begin with, the two points defining the extremes of the path were identified, starting from Colchester Castle and ending at the entrance to the Culver Square Shopping Centre, in order to cross a good part of the city section and diversify the areas and streets encountered.

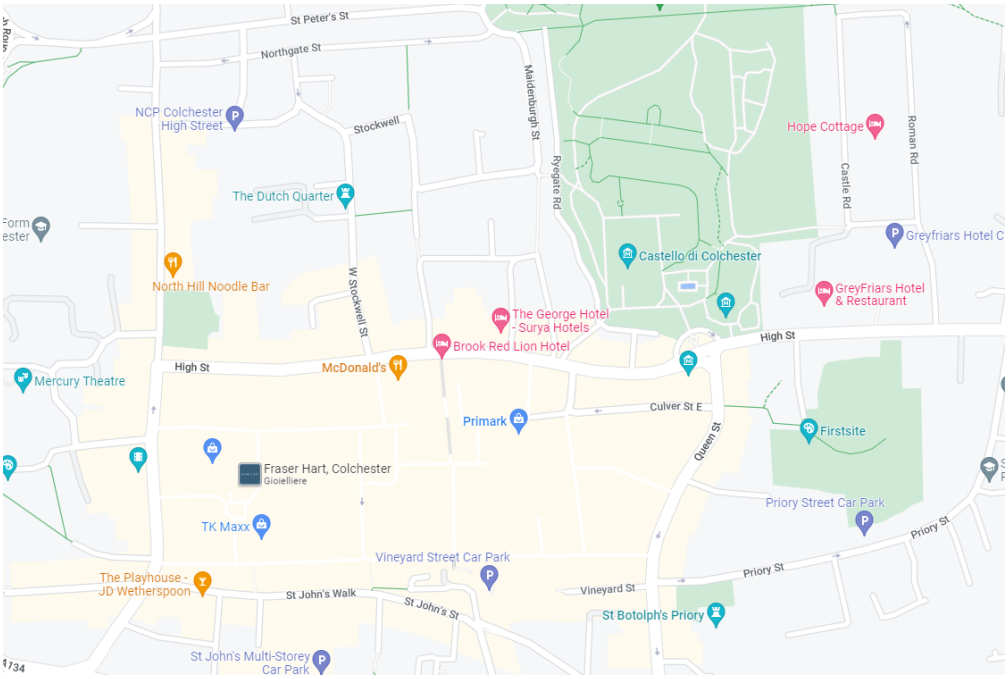


Figure 5.9: Map preview of the Colchester sections used in the project

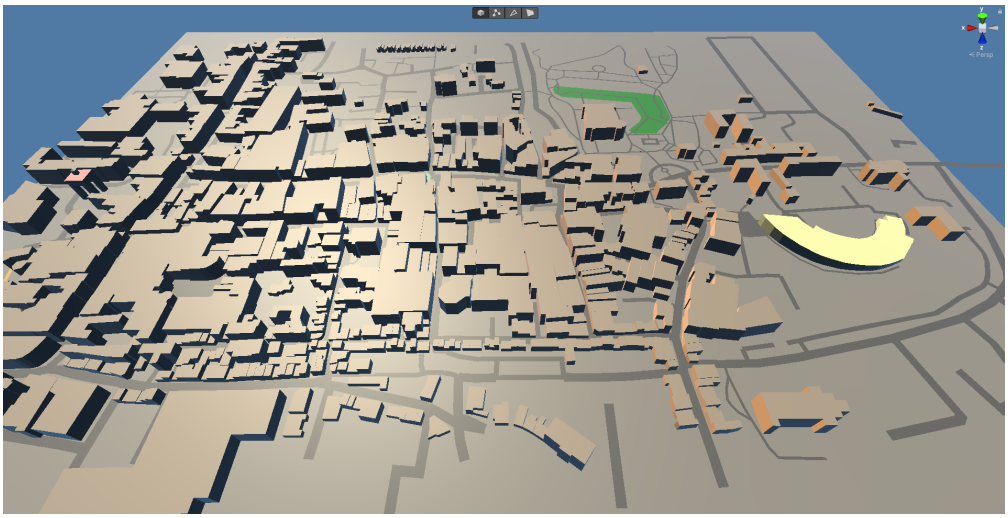


Figure 5.10: Virtual Colchester overview as imported without textures.

The proposed virtual environment is thus composed as follows:

- A system of roadways representing the areas that can be traversed by the user during the simulations, which cannot be crossed by pedestrians and are distinguishing for the nodes that will form the graph on which search

algorithms are applied.

- A horizontal plane covering the entire imported section, each area belonging to this plane represents pedestrian areas and where the user cannot walk during sessions.
- A set of buildings that constitute the obstacles on the map.

Overall, the proposed area has a width of about 1 km and a height of about 700 metres. As a size for the grid cells, 1 m was chosen, in order to have a good granularity and at the same time not to increase the complexity too much, allowing several path computations in a short time.

Taking the values of the weights w_i obtained from the last experiments carried out in the first reproduction of a city scene, the TOBS_A* algorithm is applied to the section of Colchester focusing on the area near the castle, used as a starting point, to examine its behaviour. Figure 5.11 shows the path resulting from the algorithm applied using $w_1 = 0.3$, $w_2 = 1$ and $w_3 = 0.1$ as weights. It can be seen in this case how the path tends to prefer a wider route than the actual shortest one to reach the High Street and proceed, effectively lengthening the path considerably.

It is easy to see how this problem originates from the actual contribution that its components make to the heuristic function used. Although h_1, h_2 and h_3 contribute to each other as desired, their weight is excessive compared to the weight of the basic component h_0, characterising the most promising direction to reach the endpoint.

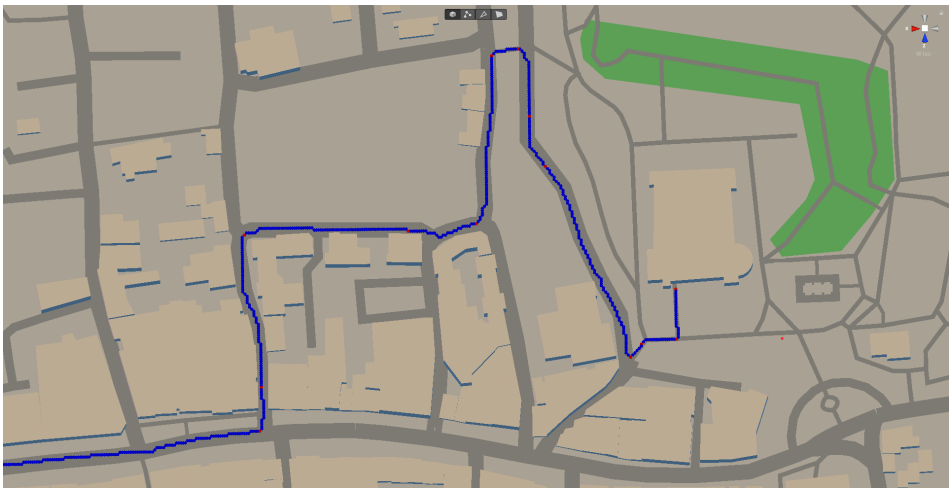


Figure 5.11: Portion of the resulting path with $w_1 = 0.3, w_2 = 1, w_3 = 0.1$.

By performing an initial proportional reduction of the w_i parameters, and then further tuning them, it was possible to find a more suitable configuration for this

type of environment. In this example, the objective is to define a route that favours the crossing of High Street, represented by the long horizontal street in the foreshortened view, while passing through streets that possess 3 or 4 nodes in width. After a number of attempts, the values for the parameters that meet the defined requirements that have been found are:

- $w_1 = 0.1$
- $w_2 = 0.2$
- $w_3 = 0.5$

Below there is the same portion of the map with the path found using these weights for the contributions of the h function.

Figure 5.12: Portion of the resulting path with $w_1 = 0.1, w_2 = 0.5, w_3 = 0.2$.

To emphasise the results obtained by the algorithm, two further points of interest were identified to calculate an orientation path. The starting point was placed at the Firstsite art gallery, located just below Colchester Castle, and the destination was placed at another art gallery located in the northwest of the virtual environment, the Colchester Gallery. Figure 5.13 shows the entire path defined using the TOBS_A* algorithm and the weights obtained from the previous experiment. The pattern of the path reflects what is desired, proceeding along High Street as far as possible, then turning into North Hill and proceeding to the Colchester Gallery intersection.

Figure 5.14 below shows the shortest path from the same starting point to the same destination found using the basic A* algorithm. It is evident that even in this case the route tends to remain as close as possible to the line connecting the source to the destination location, increasing the number of turns taken in the trip.

Figure 5.13: Path defined with TOBS_A* from Firstsite art gallery to Colchester Gallery.

Figure 5.14: Same path defined with A* algorithm.

5.1.5 Navigation tests

For each path, the distance is stored as an accessible parameter on the corresponding game object. By referring to the user's average speed of travel, the average completion time for each path is recorded as well. A number of paths have been defined in order to document tests in which they are reproduced from the start to the destination, measuring the time taken and the distance covered by the overall movement of the user. The objective is to verify that these measurements coming from a user attempting to reproduce the path by knowing the route actually match

the distance determined by the path finding algorithm and the calculated average time.

The paths defined for these tests are listed below, in which are included the two presented earlier in this chapter.

Path	Starting Point	End Point	Distance	Avg Time
1	Colchester Castle	Culver Square	**739** m	**185** s
2	Firstsite Gallery	Colchester Gallery	**987** m	**247** s
3	Head St.	Herrick House	**703** m	**176** s
4	Water Tower	Spiritualist Church	**1175** m	**294** s
5	Trinity St Church	ODEON Cinema	**276** m	**69** s

Table 5.1: List of paths defined for tests and data analysis.

By importing the virtual city using procedural elaboration of a section of the real map of Colchester, designed to have dimensions that refer to Unity units as to metres, it was possible to create distance-accurate paths. The following figure shows path 1 reproduced on Google Maps with an indication of the corresponding distance.

For each path, five tests were conducted with five different people. In each test, the player is asked to try to reach the destination by following directions given directly to the user, thus reproducing the case where the path is known to the player. In addition to the directions, the itinerary map is kept visible throughout the test in order to have a clearer view of the route to be taken.

In order to take into account the impact of the familiarity of controls that improves as the test progresses, the different paths are proposed to the subjects in random order. Control of the game avatar is performed via keyboard and mouse, running tests directly on a desktop PC, using Unity's editor demo system.

The defined test paths are now briefly described:

- The first path has its starting point in front of the Colchester Castle, while its destination is the Culver Square shopping centre. It is a medium-length path that extends mostly along the High St. main street.

- The second path is longer and starts at the Firstsite Gallery and proceeds to another art gallery located in the northwest area of the Colchester section used. It is an easy path as it only involves two turnings, travelling through the entire High St.

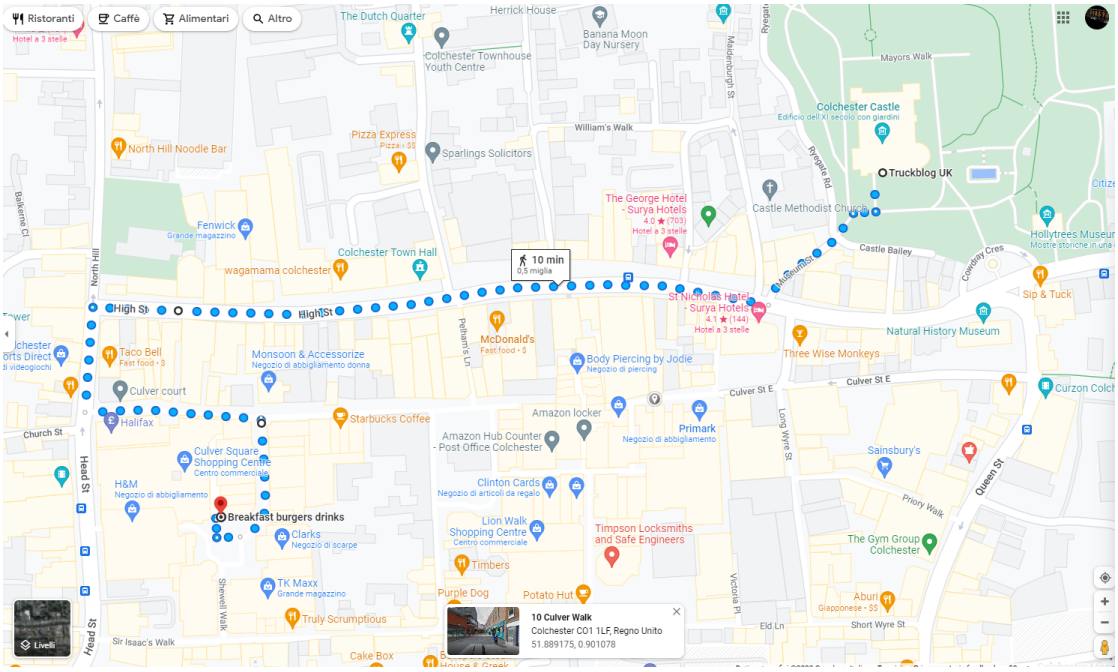


Figure 5.15: Path 1 represented in a Google Maps itinerary.

- In the third path, the user starts by coming from a street called Head St. located to the southwest area of the map, and then proceeds until a residential building further north. It is of medium length and a little more complicated in its routing.
- The fourth path is the longest of the defined ones, starting at the Water Tower to the west of the used section and, traversing the entire southernmost area of the map, reaching a church located in the extreme south-east of the explorable area.
- Finally, the fifth path is the shortest and explores a small area near Culver Square, starting from a church and arriving at a cinema located in Head St.

Path 1

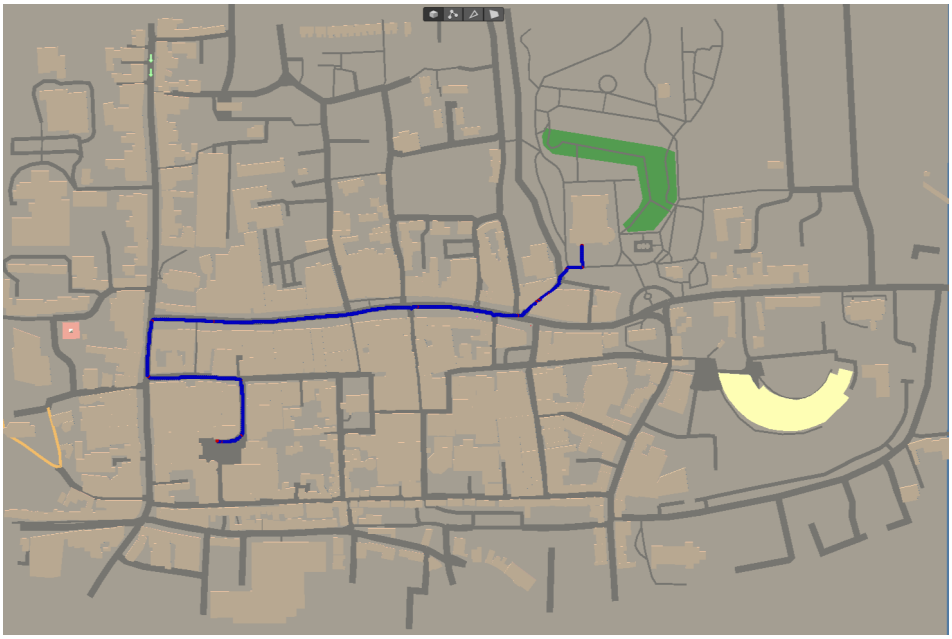


Figure 5.16: Path 1 overview.

Subject	Distance Covered	Completion Time	e(Distance)	e(Time)
A	738 m	192 s	-1 m	+7 s
B	744 m	198 s	+5 m	+13 s
C	735 m	204 s	-4 m	+19 s
D	738 m	196 s	-1 m	+11 s
E	743 m	194 s	+4 m	+9 s

Table 5.2: Test data from simulations on path 1.

Path 2

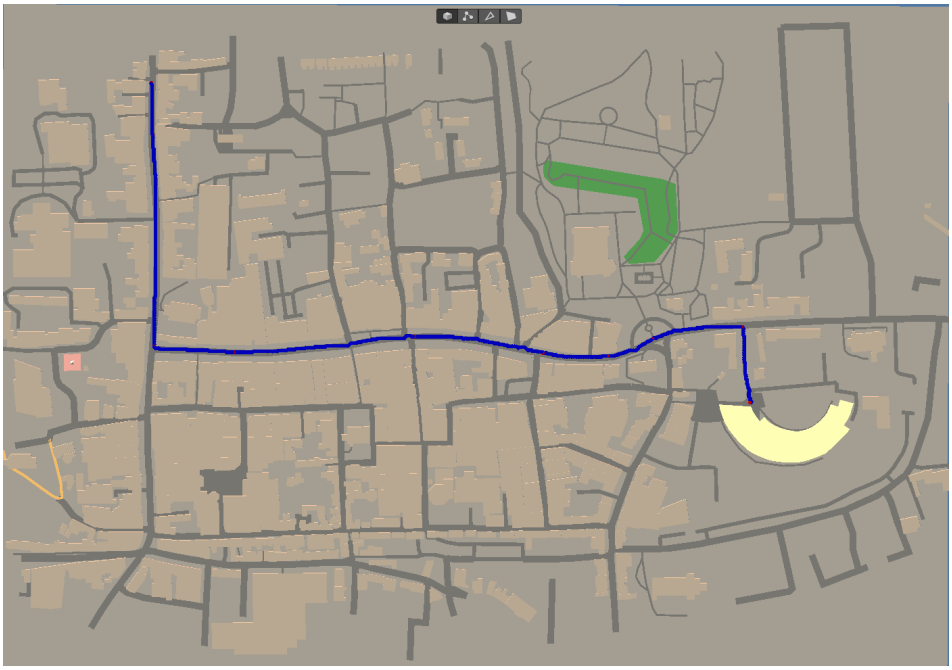


Figure 5.17: Path 2 overview.

Subject	Distance Covered	Completion Time	e(Distance)	e(Time)
A	997 m	250 s	+10 m	+3 s
B	993 m	265 s	+6 m	+18 s
C	985 m	250 s	-2 m	+3 s
D	990 m	253 s	+3 m	+6 s
E	993 m	257 s	+6 m	+10 s

Table 5.3: Test data from simulations on path 2.

Path 3

Figure 5.18: Path 3 overview.

Subject	Distance Covered	Completion Time	e(Distance)	e(Time)
A	698 m	175 s	-5 m	-1 s
B	714 m	191 s	+11 m	+15 s
C	702 m	180 s	-1 m	+4 s
D	710 m	188 s	+7 m	+12 s
E	699 m	184 s	-4 m	+8 s

Table 5.4: Test data from simulations on path 3.

Path 4

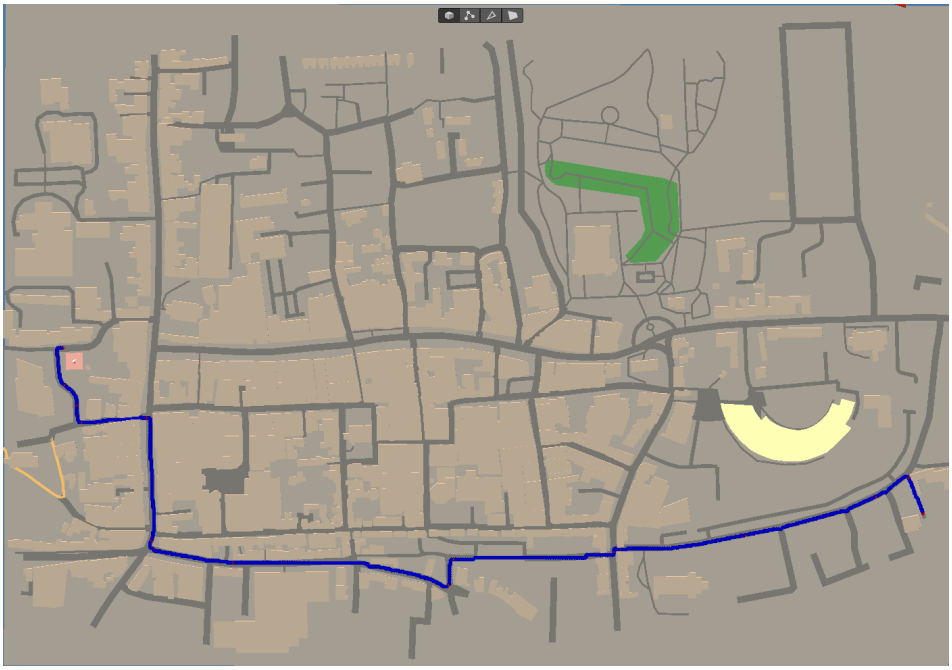


Figure 5.19: Path 4 overview.

Subject	Distance Covered	Completion Time	e(Distance)	e(Time)
A	1192 m	304 s	+17 m	+10 s
B	1216 m	312 s	+41 m	+18 s
C	1200 m	312 s	+25 m	+18 s
D	1198 m	305 s	+23 m	+11 s
E	1188 m	306 s	+13 m	+12 s

Table 5.5: Test data from simulations on path 4.

Path 5

Figure 5.20: Path 5 overview.

Subject	Distance Covered	Completion Time	e(Distance)	e(Time)
A	282 m	74 s	+6 m	+5 s
B	274 m	70 s	-2 m	+1 s
C	276 m	79 s	+0 m	+10 s
D	285 m	78 s	+9 m	+9 s
E	279 m	74 s	+3 m	+5 s

Table 5.6: Test data from simulations on path 5.

From the tests conducted, it emerges that the distance travelled by the subjects tends to be very close to that predicted by the path calculation, with an error $e(Distance)$ often lower than $10m$, and in any case always less than 5%. The time taken to complete the tasks is always slightly higher than the pre-calculated time, also due to the fact that the resulting average speed of the users during simulations is a bit lower than that used in the calculation, considering minor moments of stops during the journey.

In general, the results obtained from these tests are satisfactory; especially for the expected travel time, the parameter of each path is valid as a lower bound for the evaluation of the exercise.

As far as distance is concerned, it too may represent a lower limit for user performance in the platform, but this is not intended to be a real objective for the sessions, as it alone does not provide an accurate indication of the path travelled.

Chapter 6

Conclusions

In this thesis project, the aim was to create an easy-to-use tool that would allow a quick orientation path to be defined for the different phases of rehabilitation treatments for Alzheimer's patients. Each path had to be a virtual object within a Unity scene, described by a multitude of waypoints and characterised by certain parameters that could be used in the learning and evaluation phases.

The results obtained through this thesis work are satisfactory, achieving the desired objective. By varying the parameters defined for the heuristic function used by the path search algorithm by small amounts, it is possible to identify orientation paths that meet the set requirements of realism and behaviour from any starting point and destination. Thanks to this tool, it was possible to define a method for creating an abstract representation of the map, which can also be used for various other purposes, such as the insertion of NPCs in the virtual environment. The representation of paths via a set of waypoints allows the platform system to track progress, provide directions, recognise any loss of orientation and guide the user back to the correct steps.

The tool for defining paths proved to be optimal in offline processing, but began to have too high computation times in cases where the map size was too large compared to its granularity (number of nodes $> 4 \cdot 10^6$).

As for the path search performed at runtime, the algorithm based on A* and line-of-sight performs well, taking advantage of Unity's threading and coroutines mechanism. Here again, however, an excessively large size ratio between the map and the cells leads to a non-tolerable slowdown of the online search algorithm.

The pedestrian management tool was also realised in accordance with the intended objectives; it is possible to easily manipulate the defined parameters concerning the actual spawning of NPCs in the map and those concerning the behaviour they have, in order to adapt to the desired virtual environment. Finally, multiple functions have been defined to capture information on the patient's advancement and progress in the simulation, so that it can be processed by the

virtual assistant or the system in general for when the actual treatment phases are implemented.

As for further developments, possible insights or improvements can be discussed divided into different parts that constitute this thesis.

- Concerning the definition of the orientation paths, the accuracy of the two distance parameters, minimum and maximum, which characterise the individual checkpoints of the path, can be improved. Where the criteria concerning the minimum distance parameter to a checkpoint, used to define the overcoming of the given checkpoint, are adequate in their operation, those concerning the maximum distance and consequently the loss of orientation should be explored in depth. Once the phases of the treatments are specified in their methods and constitutions, better criteria for the loss and recovery of orientation could be defined. Therefore, considerations should be made that consider not only the distance between the user and the last checkpoint passed and the distance between the user and the next checkpoint to be reached, but also the direction most recently taken by the patient, the direction in which they are currently facing and/or the time elapsed since the last checkpoint was passed.

- The online path search uses algorithms based on the grid representation of the map and consequently implements processing with a certain degree of complexity, which would not be necessary to design a path based merely on the directions to be taken to reach a certain location. A pre-calculated waypoint system for the map could easily handle an environment consisting mainly of roads and crossings; leaving the online part to a simple search algorithm based on a small number of waypoints describing the roads on the map could greatly simplify calculations performed at runtime compared to the current methods.

- Once the virtual environment is enriched with more diversified elements, other types of pedestrian behaviour can be added. Following the system of primary and secondary behaviours, other types of NPC actions can be added so that they interact more with the environment, e.g. crossing the street at crosswalks, observing elements of interest in the city, shop windows, signs. Optionally also add actions concerning interaction with the user, such as a greeting or some sentence spoken when the player and the NPC come to meet.

This study once again demonstrates the potential that path finding techniques can bring to the field of video games and automation. The versatility of the most common algorithms allows the problem to be adapted to more or less complex situations. Combinations of the advantages and needs of these algorithms, such as pre-computation of certain search elements, processing of different representations of the analysed environment, and hierarchisation of tasks make it possible to

implement path finding systems as accurate as necessary in the field of application. The simplicity and different functionalities of Unity then made it possible to achieve all the set goals by implementing functions and scripts that are easily accessible, debuggable, and refinable. It also made it possible to quickly create virtual demo environments for testing and analysing results, making use of Unity's Gizmos functions that allow data and abstractions to be visualised directly in the scene in the Unity editor.

Bibliography

[1] Henry Fuchs and Gary Bishop. *Research directions in virtual environments*. 1992 (cit. on p. 1).

[2] Carolina Cruz-Neira. «Virtual reality overview». In: *Siggraph*. Vol. 93. 23. 1993, p. 2 (cit. on p. 1).

[3] Asmaa Saeed Alqahtani, Lamya Foaud Daghestani, and Lamiaa Fattouh Ibrahim. «Environments and system types of virtual reality technology in STEM: A survey». In: *International Journal of Advanced Computer Science and Applications (IJACSA)* 8.6 (2017) (cit. on p. 1).

[4] Shi Cao. «Virtual Reality Applications in Rehabilitation». In: vol. 9731. July 2016, pp. 3–10. ISBN: 978-3-319-39509-8 (cit. on p. 2).

[5] Riva G. «Applications of virtual environments in medicine». In: *Methods Inf Med* 42.5 (2003) (cit. on p. 2).

[6] Freeman D. «Studying and treating schizophrenia using virtual reality: a new paradigm». In: *Schizophr Bull* 34.4 (July 2008) (cit. on p. 3).

[7] Maria Schultheis and Albert Rizzo. «The application of virtual reality technology in rehabilitation». In: *Rehabilitation Psychology* 46 (Aug. 2001), pp. 296–311 (cit. on p. 3).

[8] Lee Ae Young Oh Eungseok. «Mild Cognitive Impairment». In: *J Korean Neurol Assoc* 34.3 (2016), pp. 167–175 (cit. on p. 5).

[9] Charles Christiansen, Beatriz Abreu, Kenneth Ottenbacher, Kenneth Huffman, Brent Masel, and Robert Culpepper. «Task performance in virtual environments used for cognitive rehabilitation after traumatic brain injury». In: *Archives of Physical Medicine and Rehabilitation* 79.8 (1998), pp. 888–892. ISSN: 0003-9993 (cit. on p. 5).

[10] Mónica S Cameirão, Sergi Bermúdez i Badia, Esther Duarte, Antonio Frisoli, and Paul F M J Verschure. «The combined impact of virtual reality neurorehabilitation and its interfaces on upper extremity functional recovery in patients with chronic stroke». In: *Stroke* 43.10 (Oct. 2012), pp. 2720–2728. ISSN: 0039-2499 (cit. on p. 5).

[11] Kim JH Kim O Pang Y. «The effectiveness of virtual reality for people with mild cognitive impairment or dementia: a meta-analysis». In: *BMC Psychiatry* 19.219 (July 2019) (cit. on p. 6).

[12] Clay F et al. «Use of Immersive Virtual Reality in the Assessment and Treatment of Alzheimer's Disease: A Systematic Review». In: *J Alzheimers Dis.* 75 (May 2020), pp. 23–43 (cit. on p. 6).

[13] Moussavi Z White PJ. «Neurocognitive Treatment for a Patient with Alzheimer's Disease Using a Virtual Reality Navigational Environment». In: *J Exp Neurosci* 10 (Nov. 2016), pp. 129–135 (cit. on p. 7).

[14] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. «Theta*: Any-Angle Path Planning on Grids». In: *J. Artif. Intell. Res. (JAIR)* 39 (Jan. 2014) (cit. on p. 7).

[15] Geethu Elizebeth Mathew. «Direction Based Heuristic for Pathfinding in Video Games». In: *Procedia Computer Science* 47 (2015), pp. 262–271 (cit. on p. 8).

[16] Xiao Cui and Hao Shi. «Direction Oriented Pathfinding In Video Games». In: *International Journal of Artificial Intelligence I& Applications* 2 (Oct. 2011) (cit. on p. 8).

[17] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. English. MIT Press, May 2005 (cit. on p. 9).

[18] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. «A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games». In: *Int. J. Comput. Games Technol.* 2015 (Jan. 2015) (cit. on p. 10).

[19] Peter Yap. «Grid-Based Path-Finding». In: *Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*. AI '02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 44–55 (cit. on p. 10).

[20] Suping Zhao, Bruno Siciliano, Zhanxia Zhu, Alejandro Gutierrez-Giles, and Jianjun Luo. «MULTI-WAYPOINT-BASED PATH PLANNING FOR FREE-FLOATING SPACE ROBOTS». In: *International Journal of Robotics and Automation* 34 (Jan. 2019) (cit. on p. 12).

[21] Adi Botea, Bruno Bouzy, Michael Buro, Christian Bauckhage, and Dana S. Nau. «Pathfinding in Games». In: *Artificial and Computational Intelligence in Games*. 2013 (cit. on p. 13).

[22] William Lee and Ramon Lawrence. *Fast Grid-based Path Finding for Video Games* (cit. on p. 14).

[23] Nathan R. Sturtevant and Michael Buro. «Improving Collaborative Pathfinding Using Map Abstraction». In: *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference*. The AAAI Press, 2006, pp. 80–85 (cit. on p. 14).

[24] Yang Xue and Jian-Qiao Sun. «Solving the Path Planning Problem in Mobile Robotics with the Multi-Objective Evolutionary Algorithm». In: *Applied Sciences* 8.9 (2018) (cit. on p. 15).

[25] Yoppy Sazaki, Anggina Primanita, and Muhammad Syahroyni. «Pathfinding car racing game using dynamic pathfinding algorithm and algorithm A». In: July 2017, pp. 164–169 (cit. on p. 15).

[26] Frank Steinicke, Timo Ropinski, and Klaus H. Hinrichs. «A generic virtual reality software system's architecture and application». In: *ICAT '05*. 2005 (cit. on p. 20).

[27] Michael Lewis and Jeffrey Jacobson. «Game engines in scientific research». In: *Communications of the ACM* 45 (Jan. 2002), pp. 27–31 (cit. on p. 20).

[28] Tuukka Takala. «A Toolkit for Virtual Reality Software Development - Investigating Challenges, Developers, and Users». PhD thesis. Jan. 2017 (cit. on p. 21).

[29] Daniel Foead, Alifio Ghifari, Marchel Kusuma, Novita Hanafiah, and Eric Gunawan. «A Systematic Literature Review of A* Pathfinding». In: *Procedia Computer Science* 179 (Jan. 2021), pp. 507–514 (cit. on p. 33).

[30] Rashmi Ballamajalu. «Turn and Orientation Sensitive A* for Autonomous Vehicles in Intelligent Material Handling Systems». Rochester Institute of Technology, 2020 (cit. on p. 34).