TECHNICAL UNIVERSITY OF CRETE, GREECE

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Simulation of Natural Disaster Response on Tangible Geographic Information Systems

Ioannis Brellas

Chania, April 2019

Πολυτεχνείο Κρήτης

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Προσομοίωση Αντιμετώπισης Φυσικών Καταστροφών σε Απτά Γεωγραφικά Πληροφοριακά Συστήματα

Ιωάννης Μπρέλλας

Χανιά, Απρίλιος 2019

Abstract

Nowadays, Natural Hazards occur on a daily basis and typically require immediate human intervention, in order to prevent further disasters or casualties. In most cases, the scene conditions can be unpredictable or can change at extremely fast rates, due to factors that cannot be controlled or predicted. Cases like these consist ideal testbeds for modern, digital modeling and planning tools, as their use may enhance rescue teams' actions and efficiency. In this thesis, we present the use of a tangible Geographic Information System (GIS) as a support tool that can be used to assist search and rescue teams in preventing or stopping natural disasters. Specifically, a box filled with sand serves as a malleable terrain model that can be used to formulate any type of landscape. The physical form of the model is then digitized through a Kinect sensor and a combination of a 3D model and a GIS is implemented. On this 3D model of the tangible GIS we integrate a wildfire spread model and pathfinding algorithms, along with object detection techniques. The end result is a robust simulation of a real life firefighting scenario, where the 3D terrain model, the fire spread simulation and the minimum cost path are all projected on the sand's surface and the user can alter the parameters of the simulation in real time. The entire project has been implemented withing the Unity3D game engine and can be used as an executable application on all supported platforms. The main advantage of the proposed scheme is its real time aspect, as well as its adaptability to any kind of natural disaster and its ease of use, making it accessible to groups of no expertise on digital technologies.

Περίληψη

Στις μέρες μας, οι φυσικές καταστροφές παρουσιάζονται σε καθημερινή βάση και συνήθως απαιτείται άμεση ανθρώπινη επέμβαση, ώστε να αποφευχθούν περαιτέρω ζημιές ή ανθρώπινες απώλειες. Στις περισσότερες περιπτώσεις, οι συνθήκες σε ένα τέτοιο συμβάν μπορεί να είναι απρόβλεπτες ή μπορεί να αλλάζουν με υπερβολικά γρήγορους ρυθμούς, λόγω παραγόντων που δεν μπορούν να ρυθμιστούν ή να προβλεφθούν. Τέτοια φαινόμενα αποτελούν ιδανικές περιπτώσεις για την εφαρμογή σύγχρονων, ψηφιακών εργαλείων μοντελοποίησης και οργάνωσης σχεδίων ή στρατηγικών, καθώς η χρήση τους μπορεί να ενισχύσει την δράση και την αποτελεσματικότητα των ομάδων διάσωσης. Η παρούσα διπλωματική παρουσιάζει την χρήση ενός απτού Γεωγραφικού Πληροφοριακού Συστήματος (ΓΠΣ) ως ένα εργαλείο, το οποίο θα μπορούσε να βοηθήσει τις ομάδες διάσωσης στην αποφυγή ή την αντιμετώπιση φυσικών καταστροφών. Συγκεκριμένα, ένα κουτί γεμάτο με άμμο χρησιμοποιείται ως ένα εύπλαστο φυσικό μοντέλο, πάνω στο οποίο μπορεί να σχηματιστεί οποιοσδήποτε εδαφικός τύπος. Το φυσικό αυτό μοντέλο ψηφιοποιείται μέσω του αισθητήρα Κινεςτ και δημιουργείται ένας συνδυασμός ενός τρισδιάστατου μοντέλου και ενός ΓΠΣ. Πάνω σε αυτό το τρισδιάστατο μοντέλο, ενσωματώνουμε ένα μοντέλο εξάπλωσης φωτιάς και αλγορίθμους εύρεσης μονοπατιών, καθώς και τεχνικές εντοπισμού αντικειμένων. Το τελικό αποτέλεσμα είναι μια προσομοίωση ενός πραγματικού σεναρίου πυρόσβεσης, στην οποία το τρισδιάστατο μοντέλο, η εξάπλωση της φωτιάς και τα μονοπάτια ελαχίστου κόστους προβάλλονται όλα πάνω στην άμμο και ο χρήστης μπορεί να αλλάξει τις παραμέτρους τις προσομοίωσης σε πραγματικό χρόνο. Η συνολική εργασία έχει υλοποιηθεί μέσω της μηχανής παιχνιδιών Υνιτψ3Δ και μπορεί να χρησιμοποιηθεί σαν εκτελέσιμη εφαρμογή σε όλες τις υποστηριζόμενες πλατφόρμες. Τα βασικά πλεονεκτήματα του προτεινόμενου εργαλείου είναι η δυνατότητα λειτουργίας του σε πραγματικό χρόνο, η προσαρμοστικότητα του σε κάθε μορφή φυσικής καταστροφής και η ευκολία στη χρήση του, πράγμα που το κάνει προσιτό και σε μη τεχνολογικά καταρτισμένο κοινό.

Acknowledgements

First, I would like to thank my advisor Michail G. Lagoudakis for his valuable help and guidance during the course of this thesis and my professor Panagiotis Partsinevelos for his endless support and for giving the chance to do my research at his laboratory.

Next, I would like to thank my friends and members of “Senselab” team , namely Xatziparasxis Dimitris, Antonopoulos Angelos, Kavroulakis Alexandros and Trigkakis Dimitris who stood by me whne i need them and shared with me some great ideas.

My friends from Chania, Giorgos, Sotiris, Giannis, Vasilis D., Andreas, Vasilis G.and Eleni with whom i shared a lot of great moments throughout these years.

Last, but not least, I would like to thank my family for their love, support, and constant encouragement.

Contents

List of Figures

List of Algorithms

LIST OF ALGORITHMS

Chapter 1

Introduction

Nowadays, Natural Hazards occur almost on a daily basis and commonly require immediate rescue intervention, in order to prevent human casualties. In most cases, the scene conditions can be very unpredictable or can be altered in extremely fast rates due to factors that cannot be controlled by humans. Such cases, require very careful planning to approach the situation in the best possible way and prevent any additional disasters.

Natural disasters, though, are not as simple as they may seem. Each type of disaster is affected by different factors, each of which have to be taken into account when planning the corresponding response. Therefore, the search and rescue teams have to study each and every of the factors that can alter the behavior of the disaster that needs to be stopped or prevented, in order to come up with a robust plan that will result in saving as much lives and properties as possible before it is too late.

However, the disparity between the natural hazards and the vast amount of single factors that have to be studied have lead to the categorization of search and rescue teams into teams that are responsible for a single disaster rather than all of them. This way, each team has to focus only on the disaster that they are responsible for, leading in a more efficient prevention or termination of it. Although, this separation of the teams provided a better way to tackle natural disasters, it was not, yet, the best that could be done. Sometimes, the information needed to approach a hazard in the best way possible is so vast, that a single team cannot process them fast enough, resulting in failures or late responses.

This is the reason why many organizations and research teams have developed applications that can be used to assist search and rescue teams. These applications are mostly focused on providing a simulation of the ongoing or "to happen" natural disaster, based on the factors of each individual scenario. This way, teams can study the evolution of the hazard and plan a strategy on stopping or preventing the disaster. This results in faster and more effective intervention to the crisis caused by the hazard, based on the information they get from the applications.

Sometimes, though, the factors of a natural disaster can act or alter in an unpredictable pattern. Also, throughout the duration of the rescue mission, the landscape or area where the whole procedure takes place is subject to changes, caused by landslides or falling building, which cannot be taken into account beforehand. Hence, the applications that are used to plan the best response approach to the problem have to be able to simulate such unpredictable events, so that the team can have a full perspective of how to act, in case one of these events occur.

1.1 Thesis Contribution

This thesis describes the use of a tangible GIS as an alternative tool that can be used to assist search and rescue teams in preventing or stopping natural disasters. Specifically, a box filled with sand is used as the tangible GIS, on which any landscape can be modeled physically and then converted into its corresponding elevation model. The natural disaster is then simulated and a path to it is calculated. When all the calculations and procedures are completed, the simulation of the disaster and the calculated plan are projected on the sand's surface, so that the teams' members can not only study it, but also interact with the sand to alter its form and simulate the unpredictable events that can occur.

The tangible GIS is basically a combination of three components: the sandbox, a Kinect sensor and a projector unit. The sand serves are a malleable physical model of the terrain on which the disaster takes place. This means, that its shape can be altered to fit every scenario and to simulate changes that can occur on the surface. The surface of the sand and every change that appears on it are being captured by the Kinect sensor, which is responsible for digitizing the surface model and creating its elevation model. Once the corresponding GIS is implemented, it is projected through the projector unit back onto the sand.

The GIS implemented in this thesis deviates from the original implementations and it is created as a 3D model. All the information acquired through the Kinect sensor are used to form the 3D GIS model. Then the mathematical model of the disaster is attached to it and so does the path planning implementation. This way, all the aspects that are needed for both the disaster and the response to be simulated can be manipulated and studied using only one application.

So, finally, the end product of this thesis is a real-time simulation of a natural disaster and the response to it displayed on the surface of a tangible physical model made out of sand. The tangible GIS is adaptable to all categories of natural disasters and allows the users to freely manipulate the parameters that are taken into account when simulating a hazard scenario. Also, due to its real-time aspect, its form can be altered anytime throughout the simulation process, so that unpredictable events can be simulated too.

This approach is designed with a universal aspect, meaning that any search and rescue team can use this tool as an assisting application for a priori formation of response plans or education of the local population, for any natural disaster. The entire project has been implemented in Unity3D and was turned into an executable application. This way, the project can run on all the supported operating systems without Unity3D installed, as long as there is a malleable physical terrain model, like a sandbox, a Kinect sensor and a projector unit.

1.2 Thesis Outline

In Chapter 2 we present all the background information needed for this thesis. We give a brief overview of the basic components of this thesis, like the Kinect sensor, Geographic Information Systems and the Unity3D software. Furthermore, we provide basic information on wildfire modeling and algorithms that can be used for path planning and for detecting objects. In Chapter 3 we state our tangible GIS approach on simulating responses to natural disasters, referencing other tangible approaches that have been implemented to simulate disasters or other natural phenomena. In Chapter 4, we describe in detail the implementation steps of the proposed appoach, starting from the setup of the physical components (tangible GIS, Kinect), followed by all the modeling work and finishing up with the object detection integration. In Chapter 5 the results of the implemented steps are discussed, as we validate the entire system with real-world experiments. This work is concluded in Chapter 6, in which we also suggest some future plans that could extend this approach.

Chapter 2

Background

In this Chapter a brief overview of the components, the technologies and the algorithms used to implement this thesis is going to be presented, in order for the readers to further understand and comprehend each part of the development process. The topics that are going to be discussed, which are crucial for the understanding of the development process, are the Kinect Sensor, Unity3D, Wildfire Modeling, Path Finding Algorithms, Object Detection Techniques and Geographic Information Systems.

2.1 Kinect Sensor

Kinect Sensors, or Project Natal as it was code-named in its development project, is a line of motion sensing input devices produced by Microsoft Corporation. Initially, the Kinect was developed as a gaming accessory for Xbox 360 and Xbox One video game consoles and Microsoft Windows PCs. Based around a webcam-style add-on peripheral, it enabled users to control and interact with their console/computer without the need for a game controller, through a natural user interface using gestures and spoken commands. While the gaming line did not gain much traction and eventually discontinued, third-party developers and researches found several after-market uses for Kinect's advanced low-cost sensor features, as displayed in this thesis [40].

First of all it is important to examine the parts that a Kinect sensor is comprised of. A Kinect sensor includes three things [24]:

- **Color VGA video camera** - This video camera aids in facial recognition and other detection features by detecting three color components: red, green and blue.
- **Depth sensor** - An infrared projector and a monochrome CMOS (complementary metal-oxide semiconductor) sensor work together to"se" the room in 3-D regardless of the lighting conditions.
- **Multi-array microphone** - This is an array of four microphones that can isolate the voices of the players from the noise in the room. This allows the player to be a few feet away from the microphone and still use voice controls.

The Kinect sensor as well as its components are shown in Figure 2.1.

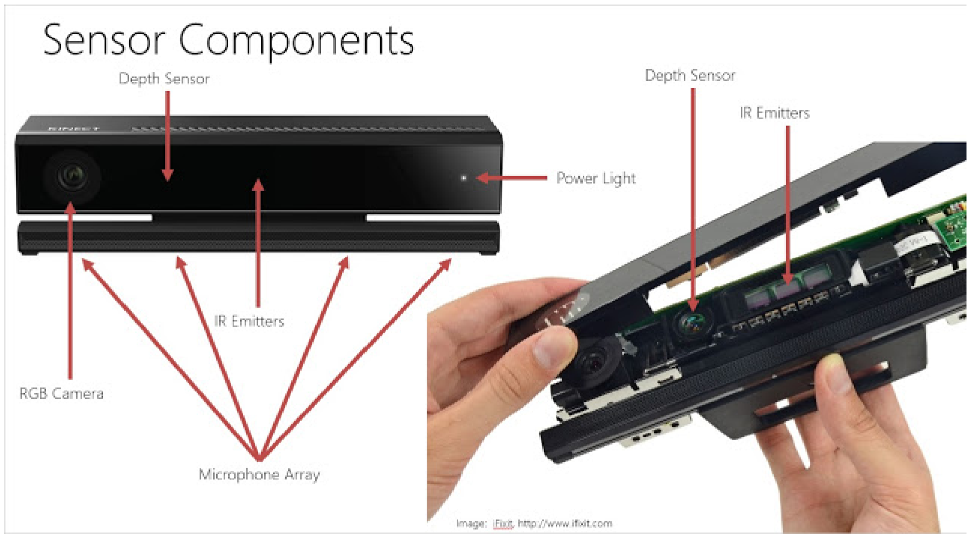


Figure 2.1: The Kinect Sensor for Xbox One [1](source)

The Kinect Sensor used in this thesis is the one released for Xbox One. It uses a wide-angle time-of-flight camera, and processes two gigabits of data per second to read its environment. This Kinect has greater accuracy with three times the fidelity over its predecessor (512x424 resolution compared to 320x240) and can track without visible light by using an active IR sensor. It has a 60% wider field of vision that can

[1]https://www.physio-pedia.com/The_emerging_role_of_Microsoft_Kinect_in_physiotherapy_rehabilitation_for_stroke_patients

detect a user up to three feet from the sensor, compared to six feet for the original Kinect, and can track up to six skeletons at once. It can also detect the player's heart rate, the facial expression, the position and orientation of 25 individual joints (including thumbs), the weight put on each limb, the speed of player movements, and track gestures performed with a standard controller. The color camera captures 1080p video that can be displayed in the same resolution (1920x1080) as the viewing screen, allowing for a broad range of scenarios. In addition to improving video communications and video analytics applications, this provides a stable input on which one can build interactive applications [41].

But the real innovation for the Kinect for Xbox One is that its driver is open source, meaning that the Kinect is susceptible to reprogramming and alterations to its supposed original functionality. Microsoft realizing this new functionality, released a SDK in order to enable not only gaming developers, but also developers that work on projects that require functionality like the one that Kinect provides. Kinect's SDK enables developers to program the functionality and the outcome of the Kinect sensors and build applications with C++, C# or Visual Basic by using the Microsoft Visual Studio [8].

Nowadays the Kinect is not part of the Xbox console anymore due to a number of factors irrelevant to this thesis. Kinect is produced and sold by Microsoft as a programmable, low cost system of sensors that has the functionality described before. That has led to a numerous third party organizations and companies, outside of the Microsoft Corporation, to use the Kinect Sensor as part of their applications. Also many research labs in spacious universities have developed many applications that are based on the Kinect sensor functionality.

Philipp Robbel of MIT combined Kinect with iRobot Create to map a room in 3D and have the robot respond to human gestures [57], while an MIT Media Lab team is working on a JavaScript extension for Google Chrome called depthJS that allows users to control the browser with hand gestures [5]. Other programmers, including Robot Locomotion Group at MIT, are using the drivers to develop a motion-controller user interface similar to the one envisioned in Minority Report [52]. Alexandre Alahi from EPFL presented a video surveillance system that combines multiple Kinect devices to track groups of people even in complete darkness [19]. In December 2010, the free public beta of HTPC software KinEmote was launched; it allows navigation of Boxee and XBMC menus using a Kinect sensor [46]. Soroush Falahati wrote an application that can be used to create stereoscopic 3D images with a Kinect sensor [25].

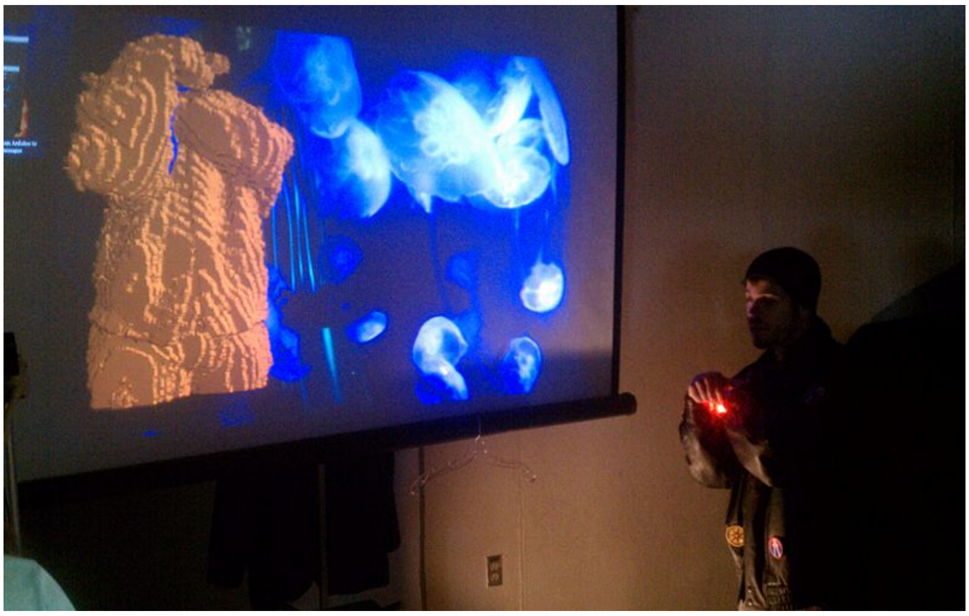


Figure 2.2: An attendant at Maker Faire 2011 wearing a jacket with bendable and pushable sensors hardwired into it, which allow the wearer to control music and video. The visualization on the left is of the wearer, and is provided through a Kinect.[2] (source)

Kinect also shows compelling potential for use in medicine. Researchers at the University of Minnesota have used Kinect to measure a range of disorder symptoms in children, creating new ways of objective evaluation to detect such conditions as autism, attention-deficit disorder and obsessive-compulsive disorder. [26] Several groups have reported using Kinect for intraoperative, review of medical imaging, allowing the surgeon to access the information without contamination. [14] This technique is already in use at Sunnybrook Health Sciences Centre in Toronto, where doctors use it to guide imaging during cancer surgery. [15] NASA's Jet Propulsion Laboratory (JPL) signed up for the Kinect for Windows Developer program in November 2013 to use the new Kinect to manipulate a robotic arm in combination with an Oculus Rift virtual reality headset, creating "the most immersive interface" the unit had built to date [37]. One of the many uses of the Kinect sensor can be see in Figure 2.2.

[2] https://en.wikipedia.org/wiki/File:Wearable_electronics_Maker_Faire_2011.jpg

So, in conclusion, the Kinect Sensor can be an excellent tool for 3D modeling, detecting and treating health and body disorders, as well as, a tool training and educating. But, the sensor could also be of good use in Geographic Information Systems (GIS). Specifically, Kinect can be a crucial factor is the development of tangible GIS - meaning GIS whose form can be altered in real time by the user - as a convenient and cheap input sensor device, since the surface of a piece of real-life landscape or an artificial substitute of it can be observed, studied or even visualized using its cameras. This opportunity will be further discussed throughout this thesis.

2.2 Unity3D

2.2.1 Unity3D Introduction

Unity3D [9] is one of the most popular cross-platform game engines created by Unity Technologies. It was first announced and released in June 2005 and since then it provides developers a very easy and friendly environment to create 2D or 3D games and applications on almost any platform they want, like mobiles, desktops, web or consoles. Unity3D gives its user full freedom on creating 3D models, interactive environments, animations, menus and developing mechanics for the game through C# or JavaScript scripts that can be attached to various parts of the game or application. Also, the user can organize the components of the implemented game/application into different scenes and environments, he/she can add lighting, audio, special effects and physics, that he/she can test or edit in real time through the editor part of the game engine. Another huge advantage that Unity3D has, is the vast amount of platforms that the implemented games or applications can be supported on (PC, Mac, PlayStation, Xbox) and the innumerable plugins that can allow the usage of external sensors or actuators.

Unity3D is defined by its component based structure, meaning that everything that is created in it, has its own specific purpose and can be used on its own or in collaboration with other components to accomplish a task. Specifically, in Unity3D every application that is created is considered a `Project`, which can consist of one or many `Scenes`. Scenes are basically the environment in which every aspect of the game or application will take place and, of course, it is a preview of the in-game look of each

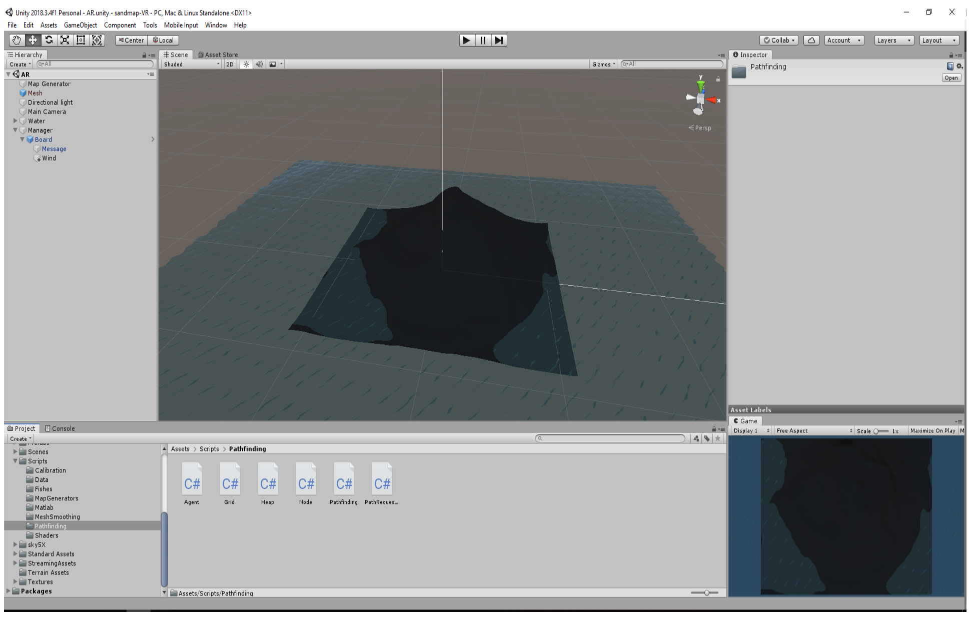


Figure 2.3: The basic layout of the Unity3D editor

level the player/user will experience when playing. In each of the scenes, the user can place 3D objects to create the game environment, which can be organized through a hierarchy system and be assigned different functions. Every one of the components that are added to the scene are called `Gameobjects`, which in turn consist of several `Components` that define their characteristics, properties and behavior. Each component can give a different look or functionality to the Gameobject and the combination of these different Gameobjects will form the final game/application. Then, the final form of the project can be build into an executable application on the desired platform and be"played" autonomously. In Figure 2.3 the basic layout of the Unity editor can be seen.

2.2.2 Gameobject Components

Some of the basic components of a Gameobject can be [39]:

- **Transform** : the most basic component of a Gameobject that defines its position in the scene, its rotation and its scale using 3D space coordinates x,y,z.

- **Mesh filter and renderer** : 3D meshes are the main graphic object primitive in Unity3D. Both the filter and the renderer are used in collaboration to display an object in the scene. The mesh filter is responsible for acquiring a mesh from the assets of the game and pass it on to the mesh renderer so that it can be rendered on the screen. The mesh renderer uses the geometry send to it by the mesh filter and renders the mesh at the position that is defined by its transform component.

- **Physics components** : the components that are responsible for the physics of the objects, meaning its motion and its behavior to collisions, by simulating physics laws. The most common of these components is the rigidbody component. This component is the one that makes the objects obey to gravity laws and simulates the behavior of the object when forces are being applied to it.Other components are the collider components of different shapes (box, sphere, capsule), which are responsible for detecting collisions between objects.

- **Renderer components** : renderer components have to do with rendering in-game and user interface elements, as well as lighting and special effects. One of the most important renderer component is the camera component. The camera component is the component that is mostly responsible for what the player will see when he/she launches the implemented application. Other common renderer components are the ones corresponding to the graphical user interface (GUI), like buttons, user interface elements and text displayed on the screen and the components responsible for the lighting of the scene.

- **Sound** : every Gameobject in the scene can also be an audio source component, which means that some of the sounds of the game or application will generate from this particular object.

- **Materials** : the material components are responsible for the textures and colors of the gameobjects of the scene. A material component determines how the gameobject will look like in a more aesthetic manner and how it will respond to the lighting of the scene. A crucial part of a material is its shader. Specifically, the shader is a specialized kind of graphical program that determines how texture and lighting information are combined to generate the pixels of the rendered object on screen.

- **Scripts** : the last but possibly the most crucial component of gameobjects (apart from their transform) is the script component. The script component is the one that defines the behavior of the object throughout the duration of the game and can trigger effects when conditions are met or an event happens in the game. The entirety of the scripts of a project is what defines the special mechanics of the game/application and makes each game unique.

2.2.3 Polygon Meshes

Unity3D, apart from a game engine, is a very good 3D modeling tool. Particularly, in Unity3D landscapes and objects' surfaces can be modeled in 3D space by using the mesh components described above and attaching to them the essential data needed for the 3D model to be shaped. Apart from the mesh component, there is another way to model a landscape environment in Unity3D. That is through a special terrain component, which allows the user to create all sorts of types of terrain, like mountains, grass or trees. The terrain component option is very straightforward, but does not give the user much freedom on its scale and size. As for meshes, a few more factors have to be defined first, before a mesh can be a viable option for 3D modeling.

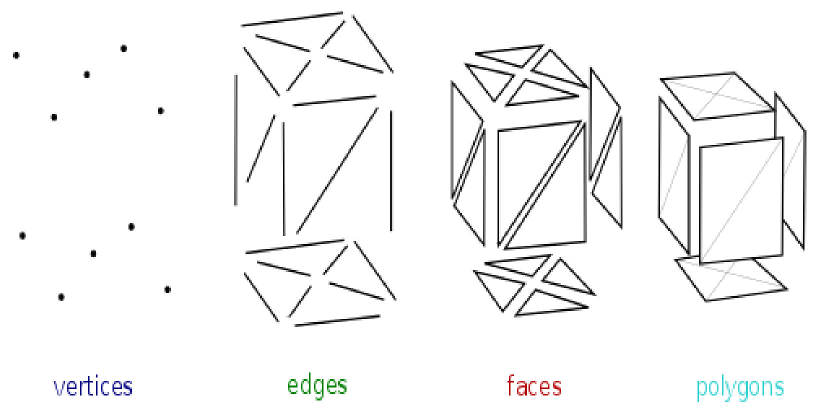


Figure 2.4: Phases of the creation of a Mesh.[3] (source)

[3]https://en.wikipedia.org/wiki/Polygon_mesh

The most common form of meshes is that of a polygon mesh [20]. Specifically, a polygon mesh consists of vertices, faces and edges and can be used to define the shape of 3D-models in computer graphics and computer animation. The vertices are a collection of positions is the 3D space defined but their x, y, z coordinates that can also carry information about color and texture coordinates. The edges are basically the connection lines between two vertices. Finally, the faces can be triangles, quadrilaterals or other convex polygons that are shaped through a closed set of edges. A visual representation of these three basic parts of a mesh can be seen in Figure 2.4. A 3D-model can simply be created by connecting multiple faces to each other. For example, a cube can be created by arranging six quad faces together which can also be defined as eight vertices positioned and connected in order to form the required faces, as shown in Figure 2.5 [7].

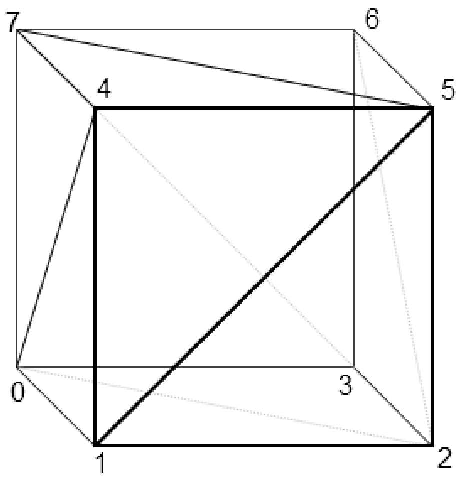


Figure 2.5: The Mesh of a Cube.[4] (source)

In Unity3D meshes [2] are defined in a similar way. All meshes in Unity3D consist only of triangles and are defined by just there three corner points or vertices. The basic info contained in a mesh class are stored in two arrays. One is a `Vector3` array where all the vertices in the 3D space are stored and the other one is an integer array where each triangle is specified using three integers that correspond to indices of the vertices array. This means that the first three elements $(0, 1, 2)$ of the integer array correspond to the first three vertices of the vertex array and define the first triangle.

However, the triangles are only enough to define the basic shape of the 3D object, but in Unity3D there are extra information that need to be displayed in order to have a full image of the concept. These extra information are about lighting and textures. The lighting corresponds to the way that the created mesh reacts to the lighting of the scene and can be automatically determined through its light renderer. Textures are

[4] http://glasnost.itcarlow.ie/~powerk/GeneralGraphicsNotes/meshes/polygon_meshes.html

essential in order to give a 3D model a more realistic and accurate representation of a real life 3D object or a landscape.

Textures in Unity3D are basically images that can be stretched and attached on objects in order to create fine details on there surfaces. When it comes to meshes, the texture image is divided in triangular areas which are then stretched in order to fit in the mesh triangle that they have to be attached. To make this work, each vertex needs to store the coordinates of the image position that will be pinned to it. These coordinates are two dimensional and scaled to the 0..1 range (0 means the bottom/left of the image and 1 means the right/top). To avoid confusion these coordinates are stored in a different array from the vertices one and there are most commonly named UV coordinates.

2.3 Geographic Information Systems

None of the methods and technologies described in the previous chapters implement a map module, which is crucial to the implementations described in this thesis. A map or a geographical representation is implemented using a Geographic Information System or, in sort, a GIS. A geographic information system (GIS) is a system designed to capture, store, manipulate, analyze, manage, and present spatial or geographic data. The key word to this technology is Geography – this means that some portion of the data is spatial, which, in other words, means that these data are is in some way referenced to locations on the earth.

GIS applications are tools that allow users to create interactive queries (user-created searches), analyze spatial information, edit data in maps, and present the results of all these operations [23]. GIS (more commonly GIScience) sometimes refer to geographic information science, the science underlying geographic concepts, applications, and systems [32].

GIS can refer to a number of different technologies, processes, techniques and methods. It is attached to many operations and has many applications related to engineering, planning, management, transport/logistics, insurance, telecommunications, and business. For that reason, GIS and location intelligence applications can be the foundation for many location-enabled services that rely on analysis and visualization [53].

GIS is a wide concept of applications and it may seem too broad to understand. So let's break down what GIS can actually do. GIS can be used as tool in both problem solving and decision making processes, as well as for visualization of data in a spatial environment. Geospatial data can be analyzed to determine (1) the location of features and relationships to other features, (2) where the most and/or least of some feature exists, (3) the density of features in a given space, (4) what is happening inside an area of interest (AOI), (5) what is happening nearby some feature or phenomenon and, lastly, (6) how a specific area has changed over time (and in what way). Some of the uses of a GIS can be seen in Figure 2.6.

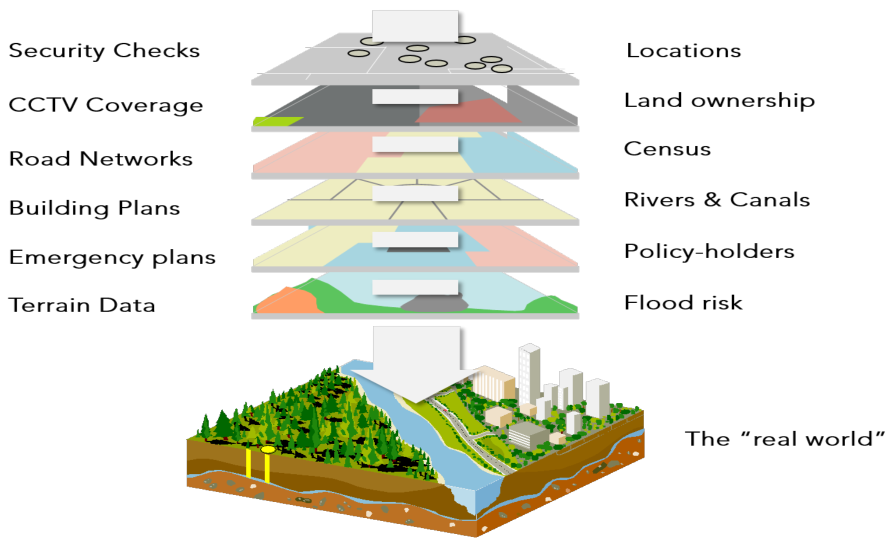


Figure 2.6: Basic concept of GIS and some examples.[5] (source)

It is important to clarify in this Chapter that GIS are not always computer applications. The first known use of the term "geographic information system" was by Roger Tomlinson in the year 1968 in his paper "A Geographic Information System for Regional Plannin". Tomlinson is also acknowledged as the"father of GIS". That project is based on multiple hand-drawn maps that represented lots of layers of the same location as part of the exploration of various areas of Canada [4]. The digital era of GIS began by the end of the 20th century, when the rapid growth in various systems had

[5]https://www.cbronline.com/what-is/gis-explained/attachment/what-is-gis/

been consolidated and standardized on relatively few platforms and users were beginning to explore viewing GIS data over the Internet, requiring data format and transfer standards. More recently, a growing number of free, open-source GIS packages run on a range of operating systems and can be customized to perform specific tasks. Increasingly geospatial data and mapping applications are being made available via the World Wide Web [30].

Modern GIS technologies use digital information, for which various digitized data creation methods are used. The most common method of data creation is digitization, where a hard copy map or survey plan is transferred into a digital medium through the use of a program, either CAD with geo-referencing capabilities or Unity. With the wide availability of ortho-rectified imagery (from satellites, aircraft, Helikites and UAVs), heads-up digitizing is becoming the main avenue through which geographic data is extracted. Heads-up digitizing involves the tracing of geographic data directly on top of the aerial imagery instead of by the traditional method of tracing the geographic form on a separate digitizing tablet.

GIS uses spatio-temporal (space-time) location as the key index variable for all other information. Just as a relational database containing text or numbers can relate many different tables using common key index variables, GIS can relate otherwise unrelated information by using location as the key index variable. The key is the location and/or extent in space-time. GIS accuracy depends upon source data, and how it is encoded to be data referenced. Land surveyors have been able to provide a high level of positional accuracy utilizing the GPS-derived positions. High-resolution digital terrain and aerial imagery, powerful computers and Web technology are changing the quality, utility, and expectations of GIS to serve society on a grand scale, but nevertheless there are other source data that affect overall GIS accuracy like paper maps, though these may be of limited use in achieving the desired accuracy [13].

GIS data represent real objects (such as roads, land use, elevation, trees, waterways, etc.) with digital data determining the mix. Real objects can be divided into two abstractions: discrete objects (e.g. houses) and continuous fields (such as rainfall amount or elevations). Traditionally, there are two broad methods used to store data in a GIS for both kinds of abstractions mapping references: raster images and vector. Points, lines, and polygons are the stuff of mapped location attribute references. A new hybrid method of storing data is that of identifying point clouds, which combine three-dimensional points with RGB information at each point, returning a "3D color imag". GIS thematic maps then are becoming more and more realistically visually descriptive of what they set out to show or determine. Data restructuring can be performed by a GIS to convert data into different formats. For example, a GIS may be used to convert a satellite image map to a vector structure by generating lines around all cells with the same classification, while determining the cell spatial relationships, such as adjacency or inclusion [56].

2.3.1 Tangible Geographic Information Systems

A special type of a Geographic Information System is this of the Tangible GIS. A Tangible GIS is based on a physical, malleable terrain model, which is scanned to get its digital elevation model and import it into a GIS. Basically, a Tangible GIS is another form of a Tangible User Interface (TUI). TUIs are user's interfaces that are based on malleable models that the user can interact with to alter and control information that were otherwise controlled digitally. This way, the user has a much clearer objective and an easier way to manipulate the data, in order to get the results that he/she desires. A flow chart of the basic concept of TUIs can be seen in Figure 2.7. Some great applications of TUIs are on construction sites, creation or regeneration of urban road networks and various simulations like wind's behavior, light and shadow interactions or even natural disasters.

So, a tangible GIS is a TUI on which a GIS has been integrated, broadening, this way, the capabilities of the system. GIS offers a set of ready-to-use tools for different types of geospatial analyses and simulations, as well as an interface for visualization. However, editing the digital elevation model on which the GIS is based on in not an easy task, since it is implemented through programs that require specialized skills (e.g. CAD). Using a tangible physical model instead of a digital one can tackle this problem entirely, since the user can experiment with different scenarios more intuitively and, of course, with greater ease. Once, the user gets the desired elevation model through the scanning of the physical model, he/she imports it into the GIS for further analyzing [44].

Then, the final step to complete the tangible GIS is to present the final form of the GIS to the user. This, can either be implemented digitally or"physicall". In the

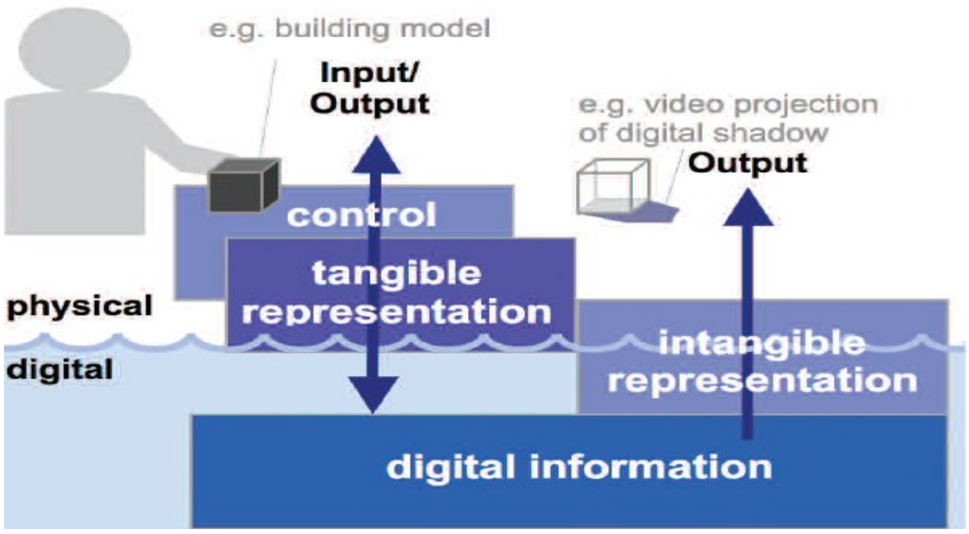


Figure 2.7: Basic concept of TUI.[6] (source)

first case, the completed GIS that derived from the terrain model can be observed and modified through a computer as a normal GIS would. In the second case, though, the final result is projected back on the physical terrain model from which the created GIS derived. This way, the geospatial information are directly attached to the part of the model that corresponds to them and the user can have an even clearer idea of the studied landscape. Also, the user can easily and directly modify the GIS or correct any faults that might have occurred during the process [45].

Tangible GIS can be a great tool to simulate several scenarios that are landscape based, like wildfires, floods, hydraulic erosion or earthquakes. Also, its ease of use enables the use of it as an educational tool for children, making it an excellent option for teaching different geography-like lessons, while its tight attachment to geospatial data can make it a essential tool for search and rescue teams.

[6]https://cacm.acm.org/magazines/2008/6/5393-the-tangible-user-interface-and-its-evolution/abstract

2.4 Wildfire Modeling

Wildfire Modeling [38] has been an area of study for computational science since 1940 and aims in understanding and predicting the fire behavior through a mathematical model. The ultimate goal of the wildfire modeling, though, is to aid in fire suppression policies. These suppression policies are separated into to categories: the preventive and the operation ones. The first one, as the name suggests, aims in trying to prevent fire bursts through organizing the available resources and constructing fire barriers that will block its spread. The operational policies are applied once a fire breaks out, in order to organize the allocation of the defense mechanisms and the evacuation of nearby residences. In both cases, a mathematical model that could predict the behavior of the fire could benefit the firefighting procedures greatly [16].

A fire propagation model takes into account many information that has to do with specific characteristics of the terrain that the fire will propagate on, as well as the weather conditions. A few of these conditions are the wind's direction and speed, the fuel type (type of vegetation) and humidity, the fuel density (vegetation thickness) and the landscape topography (slope and natural barriers). All these information have to be simulated in order for the fire propagation simulation to feel as realistic as possible, so that in turn it can be as relevant and helpful as possible in assisting in wildfires extermination [29].

Throughout the years, there have been various models that aimed on offering a solution to this problem, but one of the most important among them was Rothermel's work [49, 50]. Rothermel achieved, through experimental work, the identification of the dynamic equations that could characterize the maximum fire spread rate. Rothermel's work was enough to ignite a variety of new approaches aiming to use the aforementioned equations in order to either enhance previous approaches on fire spread modeling or invent new methods that could simulate a fire propagation more accurately.

These approaches on fire propagation modeling, old or new, can be categorized into two types according to their spatial representation. The first type of models are based on continuous planes assuming that the fire front travels on a continuous landscape and advances in a elliptical shape [48]. In this situation, the wildfire is basically considered a wave that spreads based on a system of partial differential equations.

Some of these approaches are FARsite [27], AEGIS [35] and basically most of the approaches that are based on Huygen's principle [18] on fire growth. The continuous based models can provide a quite accurate fire spread simulation but can prove to be very computationally demanding. The second type are the models that are based on grids [43]. In these approaches, the landscape is translated into a two-dimensional set of pixels and the fire behavior is calculated for each pixel individually. This makes these models much simpler and computationally faster than the continuous ones. FlamMap [28] is a very good example of such a model.

Grid-based models can be further categorized into two subcategories. The first category is based on the Bond-Percolation method, which basically means that the fire spread is modeled through historical data [36]. Although, this approach can simulate the shape of the burned area very efficiently, it cannot describe the dynamics of the fire spread [60]. Here is where the second type of grid-based models, the cellular automata based models, steps in to resolve this problem. Specifically, in a cellular automata fire spread model, the landscape is considered a finite grid, the cells of which are described by several variables and their current discrete state. As time progresses, the state of each cell changes according to a set of rules that take into account the current state of the cell and its neighboring ones. Cellular automata have proven to be a very powerful tool when it comes to predicting complex macroscopic dynamics, such as fire spread dynamics, and can be perfectly combined with geographical information systems (GIS) [54].

2.5 Path Finding

Path-finding is an important problem that has been studied throught the past and recent years and many algorithms have been implemented for it. These algorithms have many applications, including transportation routing, robot planning, military simulations, and computer games. The Path-finding procedure involves analyzing a map to find the "best" cost of traveling from one point to another. This best cost can be evaluated as a multi-valued function and use criteria such as the shortest path, least-cost path, safest path, etc. For many computer games or game-like applications this is an expensive calculation, made more difficult by the limited percentage of cycles that are devoted to AI processing [58].

Typically, a grid is superimposed over a region, and a graph search is used to find the best path. The most common approach in games is to conduct path-finding on a (rectangular) tile grid. Then, the cost to travel to each tile is defined as a positive weight. The most common path-finding algorithm used is A*. A few games use IDA* (Iterative Deepening A*), which avoids A*'s memory overhead usually at the cost of a slower search. It is worth noting, though, that there are many machine learning, reinforced learning and deep learning algorithms that could be applied on grid bases game-like applications and could result is the same or sometimes better results than graph-search based algorithms.

Path-finding in computer games may be conceptually easy, but for many game domains it is difficult to do well. Real-time constraints limit the resources — both time and space — that can be used for path-finding. One solution is to reduce the granularity of the grid, resulting in a smaller search space. This gives a coarser representation, which is often discernible to the user (characters may follow in contorted paths). Another solution is to cheat and have the characters move in unrealistic ways (e.g. teleporting). Of course, a third solution is to get a faster processor. Regardless, the demands for realism in games will always result in more detailed domain terrains, resulting in a finer grid and a larger search space.

Nonetheless, the two main algorithms that were tested in this approach for finding a path on a game-like application implemented in Unity3D will be discussed below.

2.5.1 A* Search Algorithm

One of the most common forms of an informed best-first search is the A* search algorithm. The A* algorithm [51] is more widely-known as a search algorithm that can be applied on both tree and graph structures. However, it can be easily used to find a minimum cost path on grid based structures that can be translated to graphs or trees. The way A* evaluates the path is by combining $g(n)$, which is the cost to reach node n, and $h(n)$, which is the cost to get from node n to the target node. These two costs are combined into a new accumulative cost $f(n)$:

$$f(n) = g(n) + h(n)\,. \tag{2.1}$$

Since $g(n)$ gives the path cost from the start node to node n and $h(n)$ is the estimated cost of the cheapest path from n to the target:

$$f(n) = estimated\,cost\,of\,the\,cheapest\,solution\,through\,n\,. \tag{2.2}$$

This means that when a minimum cost path has to be evaluated all that need to be done is move from the current node to the neighboring node with the lowest value of $f(n)$. The A* algorithm, in this case, is both complete and optimal, if the heuristic function $h(n)$ admissible, meaning that it never overestimates the cost to reach the goal. This condition in this case is met since $g(n)$ is the exact cost to reach node n, which in turn leads to $f(n)$ never overestimating the true cost of the solution through b.

Algorithm 1 A* search

```
1: $queue.Add(startNode)$
2: **while** $queue$ **not** $empty$ **do**
3:   $currentNode \leftarrow gueue.Remove(Node\,with\,lowest\,f$
4:   **if** $currentNode = fireNode$ **then**
5:     $done$
6:   **end if**
7:   **for** $each\,neighbor\,of\,currentNode$ **do**
8:     $Cost \leftarrow movement\ cost\ from\ current\ to\ neightbor$
9:     **if** $cost < neighbor'sGcost$ **then**
10:      $neighbor's\ G\ cost \leftarrow Cost$
11:      $neighbor's\ H\ cost \leftarrow distance\ from\ neighbor\ to\ target$
12:      $neighbor's\ parent \leftarrow currentNode$
13:      $add\ neighbot\ Node\ to\ openSet$
14:    **end if**
15:  **end for**
16: **end while**
```

Most A* implementations use a priority queue to perform the repeated selection of the minimum (estimated) cost nodes to expand. At each step of the algorithm, the node with the lowest $f(n)$ values is removed from the queue, the f and g cost values of the neighboring nodes are updated and these nodes are added to the the queue. This procedure is repeated until the target node is reached. If each node that is studied keeps track of its predecessor, then the path can be retraced by simply starting from the target node and running through all the predecessor nodes until one of these nodes is the starting node. A pseudocode of the A* search algorithm can be seen at Algorithm 1.

One final note is that A* algorithm is optimally efficient for any given heuristic function. This means that no other optimal algorithm is guaranteed to expand fewer nodes than A*. However, the fact that A* is complete, optimal and optimally efficient does not mean that it is always the better solution when it come to finding a minimum cost path to a goal. Sometimes other algorithms can be more efficient depending on the search needs.

2.5.2 Value Iteration Algorithm

The Value Iteration (VI) algorithm [51] is one of the most popular algorithm for evaluating a solution for MDPs through the evaluation of an optimal policy. This evaluation is received through the utility of each state, which in turn will determine the optimal action the agent has to make at each state. The utility of each state is the expected utility $U^{\pi}(s)$ of the state sequences that might follow it, according to the policy π that is being executed. So, given s_t is the current state that the agent is in after executing π for t steps, the expected utility of this state is calculated as:

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi, s_0 = s\right]. \tag{2.3}$$

The true utility $U(s)$ is defined as the expected sum of discounted rewards if the agent executed an optimal utility ($U(s) = U^{\pi^\star}(s)$). This utility function ($U(s)$) determines the action that the agent will select by using the Maximum Expected Utility principle, meaning that the agent will choose the action that will maximize the expected utility of the subsequent state:

$$\pi^\star(s) = \operatorname*{argmax}_{\alpha} \sum_{s'} T(s, \alpha, s') U(s'). \tag{2.4}$$

Since, the utility of a state is the expected sum of the discounted rewards from that point onwards, which means that the utility of each state is directly connected to the utility of its neighbors. Through this observation the Bellman equation can be defined, which states that *"the utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action"* and reads:

$$U(s) = R(s) + \gamma \max_{\alpha} \sum_{s'} T(s, \alpha, s') U(s') \,. \tag{2.5}$$

The utilities of the states defined by Equation 2.3 are the unique solutions of the set of Bellman equations. The Bellman equation is the basis of the value iteration algorithm, meaning that each state corresponds to a Bellman equation and the solution of this equation is the utility of this state. So, all the n Bellman equations for the n states have to be solved simultaneously, but that is not an easy task, due to the fact that the Bellman equations are nonlinear because of the "max" operator. The solution to this problem lays on an iterative approach (thus Value *Iteration*)). Specifically, all states are given an initial random utility, which then is updated through the Bellman equation, until an equilibrium is reached (convergence). The iteration step, also called Bellman update, reads:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{\alpha} \sum_{s'} T(s, \alpha, s') U_i(s') \,, \tag{2.6}$$

where $U_i(s)$ is the utility of state s at the ith iteration. When, the equilibrium is reached, the final utilities of the states are the solutions of the Bellman equations and correspond to the optimal policy. The Value Iteration algorithm can be visualized as an information spread through the states by means of local updates. A representation of the algorithm can be seen below.

In algorithm 2, S, T, R, γ correspond to the MDP's parameters, while δ is the maximum change in the utility of any state in an iteration, ε is the maximum error allowed in the utility of any state and U, U' are the vectors of the utilities for the states in S, which initially are zero.

The algorithm converges when the condition $\delta < \varepsilon(1-\gamma)/\gamma$ is met and returns a unique set of solutions for the Bellman equations. The way VI converges can be described as a contraction, which basically means that when the algorithm is applied in two different inputs in turn, the output will get "closer and closer" in every iteration.

Algorithm 2 Value Iteration

```
1: while $\delta < \varepsilon(1-\gamma)/\gamma$ do
2:   $U \leftarrow U'$
3:   $\delta \leftarrow 0$
4:   for each state s in S do
5:     $U'(s) \leftarrow R(s) + \gamma \max_{\alpha} \sum_{s'} T(s,\alpha,s')U(s')$
6:     if $\mid U'(s) - U(s) \mid > \delta$ then
7:       $\delta \leftarrow \mid U'(s) - U(s) \mid$
8:     end if
9:   end for
10: end while
11: return $U$
```

If it is supposed that the Bellman update is an operator B that is applied simultaneously to update the utility of every state and that U_i is the vector of utilities for all the states at the i-th iteration, then the Bellman update equation (Equation 2.6) can be written as:

$$U_{i+1} \leftarrow BU_i \,. \tag{2.7}$$

Next, the distances between utility vectors can be measures through the max norm. which measures the length of a vector by the length of its biggest component:

$$||U|| = \max_{s} \mid U(s) \mid \;. \tag{2.8}$$

With this definition, the “distance” between two vectors, $||U - U'||$, is the maximum difference between any two corresponding elements. The main result of this section is the following: *Let U_i and U'_i be any two utility vectors. Then:*

$$||BU_i - U|| \leq \gamma ||U_i - U'_i|| \,. \tag{2.9}$$

This means that the Bellman update is a contraction by a factor of γ on the space of utility vectors. Hence, VI always converges to a unique solution of the Bellman equations.

In particular, U'_i in Equation 2.9 can be replaced with the true utilities U, for which $BU = U$ and the inequality below is obtained:

$$||BU_i - U|| \leq \gamma ||U_i - U|| . \tag{2.10}$$

So, if the $||U_i - U||$ is assumed to be the error in the estimate U_i, it can be observed that it is reduced by a factor of at least γ on every iteration. This means that VI converges exponentially fast and the number of iterations that are required to reach a specified error bound are strongly depending on γ. However, the error bound is sometimes overly conservative as a method of deciding the runtime of the algorithm. Due to this, a bound relating the error to the size of the Bellman update on any given iteration has to be used. From the contraction property (Equation 2.9), it can be shown that if the update is small, then the error, compared with the true utility function, is also small. Precisely:

```
1: if $||U_{i+1} - U_i|| < \varepsilon(1-\gamma)/\gamma$ then
2:     $||U_{i+1} - U|| < \varepsilon$
3: end if
```

This is the termination condition of the VI algorithm shown in Algorithm 2.

Lastly, the policy loss $||U_i^{\pi} - U||$ of the algorithm has to be studied. Specifically, the policy loss defines how well will the agent do if it takes decisions based on the utility function, compared to how well it would do if it followed the optimal policy. The policy loss of π_i is connected to the error in U_i by the following inequality:

```
1: if $||U_i - U|| < \varepsilon$ then
2:     $||U_i^{\pi} - U|| < 2\varepsilon\gamma/(1-\gamma)$
3: end if
```

In practice, it often occurs that π_i becomes optimal long before U_i has converged.

So, in conclusion, the VI algorithm converges to the correct utilities and both the error in the utility estimates if the algorithm stops at a finite number of iterations and the policy loss that results from executing the corresponding Maximum Expected Utility can be bound. Let it be noted though, that all the results depend on discounting with $\gamma < 1$. If $\gamma = 1$ and the environment contains terminal states, then a similar set of convergence results and error bounds can be derived whenever certain technical conditions are satisfied.

That makes VI a very reliable algorithm when it comes to finding minimum cost paths on grid based maps, where each state corresponds to a different position of the agent on the map and the targets have initially known utilities. Applying VI on such maps will result in the propagation of the target's utility throughout the grid and thus a full mapping of the grid. This means that when the agent is placed on a grid on which VI has already run, the agent will have a full perception of which is the best action to take, no matter which is its initial position/state, simply by checking which neighbor has the best/maximum utility. Finally, since the VI converges when the final utilities of each state are the ones that correspond to the optimal policy $\pi^\star$ and the implementation of the reward function is assumed to be optimal as well, the path that the agent will follow until it reached its target is guaranteed to be the minimum cost one.

2.6 Object Detection Techniques

Object detection [42] is a computer technology related to computer vision and image processing that deals with detecting instances of semantic objects of a certain class (such as humans, buildings, or cars) in digital images and videos. The goal of object detection is to detect all instances of objects from a known class, such as people, cars or faces in an image. Typically only a small number of instances of the object are present in the image, but there is a very large number of possible locations and scales at which they can occur and that need to somehow be explored.

The basic way the object detection algorithms work is through some form of "pose" information. These information can be as simple as the location of the object, its shape, scale and color or the extent of the object defined in terms of a bounding box. In more complex cases, the pose information contains the parameters of a linear or non-linear transformation. In Figure 2.8 an example of a pose classification and its detection among other classified objects can be seen.

Object detection techniques can be categorized into two main types, the generative ones and the discriminative ones. The first type is based on probability models both for the object and its background. These models' parameters are evaluated through training and the detection decisions are based on rations of posterior probabilities. The second type is based on a classifier that can discriminate between the images that

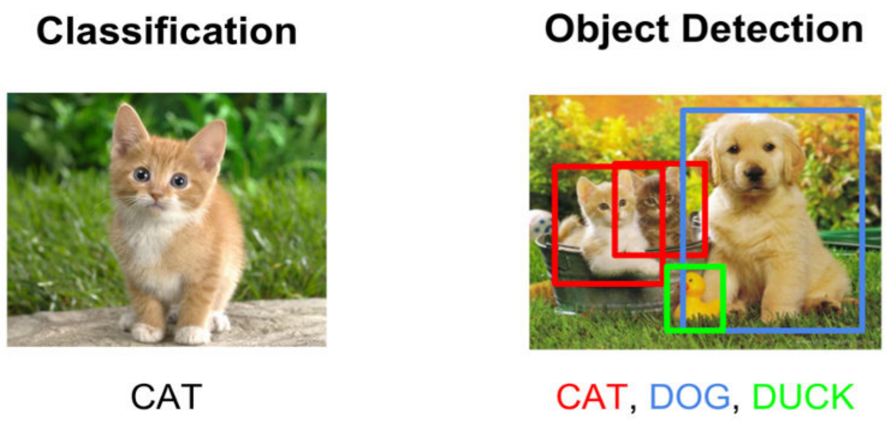


Figure 2.8: An object detection example.[7] (source)

contain the desired pose and those which don't. However, there is a third type of detection algorithms that are based on computational tools that are used to scan the entire image in order to find the desired poses [17].

Such a detection algorithm was used in this thesis, so below some of the transforms and tools used to implement that are going to be discussed.

2.6.1 Canny Edge Detector

Canny edge detector is an image processing method that uses a multi-stage algorithm to detect a wide range of edges in images. Since 1986, when John F. Canny developed it, it has been a very commonly used algorithm in many computer vision systems. Canny's intentions where to enhance the many edge detectors that already existed, so his main goals where [22]:

- *Low error rate.* The algorithm should detect all the edges without any spurious responses. This means, that the detected edges must be as close as possible to the true edges of the image.

[7] https://www.datacamp.com/community/tutorials/object-detection-guide

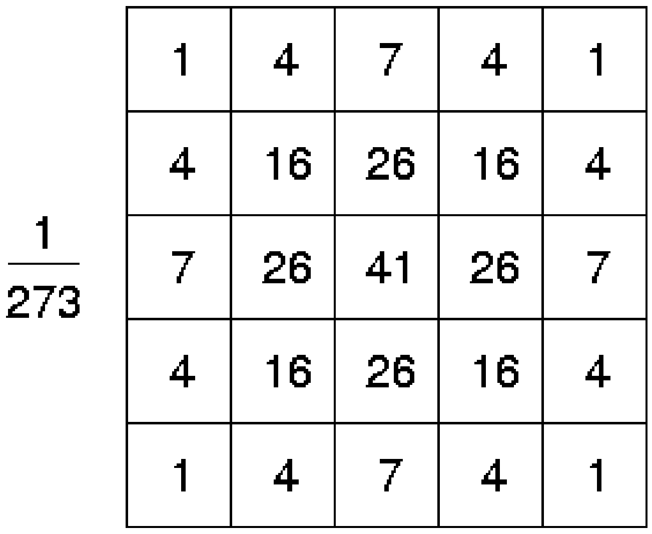
- *Edge points should be well localized.* The edges that are being detected should be as close as possible to the true edges. That is, the distance between the a point marked as an edge by the detector and the center of the true edge should be minimum.
- *Single edge point response.* The detector must return only one point for every point of the true edge. That is, the number of local maxima around the true edge should be minimum, which in turn means that the detector should not identify multiple edge pixels where only a single edge point exists.

Based on these criteria, the canny edge detector first smooths the image to eliminate any noise. It then finds the image gradient to highlight regions with high spatial derivatives. The algorithm then tracks along these regions and suppresses any pixel that is not at the maximum (nonmaximum suppression). The gradient array is now further reduced by hysteresis. Hysteresis is used to track along the remaining pixels that have not been suppressed. Hysteresis uses two thresholds and if the magnitude is below the first threshold, it is set to zero (made a non-edge). If the magnitude is above the high threshold, it is made an edge. And if the magnitude is between the two thresholds, then it is set to zero unless there is a path from this pixel to a pixel with a gradient above second threshold.

A detailed presentation of each step of the algorithm reads [33, 3]:

Step 1 The first step to implement the Canny edge detector is to filter out the noise that the image may have. The most commonly used noise canceling filter in the Gaussian one, due to its simplicity. Specifically, in the case of the Gaussian filter a simple mask has to be calculated and then applied on the image through standard convolution methods. A convolution mask is usually smaller that the actual image and the way it is applied to the image is by sliding it over the image, manipulating only a square of pixels corresponding to the size of the mask. Increasing the width of the Gaussian mask will result in a lower detector's sensitivity to noise and a slightly increased localization error. An example of such a Gaussian mask can be seen in Figure 2.9.

Step 2 After smoothing the image and eliminating the noise, the next step is to find the edge strength by taking the gradient of the image. This can be done through



$\frac{1}{273}$

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

Figure 2.9: An example of a 5x5 Gaussian mask with $\sigma = 1$.

Sobel filters. The Sobel operator performs a 2-D spatial gradient measurement on an image. Then, the approximate absolute gradient magnitude (edge strength) at each point can be found. The Sobel operator uses a pair of 3x3 convolution masks, one estimating the gradient in the x-direction (columns) and the other estimating the gradient in the y-direction (rows). An example of these operators is shown below:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \tag{2.11}$$

$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.12}$$

The magnitude and angle of the directional gradients is then calculated as:

$$\mid G \mid = \sqrt{G_x^2 + G_y^2} \tag{2.13}$$

$$\measuredangle G = \arctan G_y / G_x \tag{2.14}$$

Step 3 Once the edge direction is calculated, it has to be related to a direction that can be traced in an image. There are only four possible direction when describing the surrounding pixels of a specific pixel of the image. These are -0° (in the horizontal direction), 45° (along the positive diagonal), 90° (in the vertical direction) or 135° (along the negative diagonal). So, the direction of the edges has to be resolved into one of these four directions, depending on which one it is closer to. Therefore, the edges' directions are converted as such: the directions ranging from 0° to 22.5° and 157.5° to 180° are set to 0°, from 22.5° to 67.5° are set to 45°, from 67.5° to 112.5° are set to 90° and finally the directions ranging from 112.5° to 157.5° are set to 135°.

Step 4 The image magnitude produced results in thick edges. Ideally, the final images should have thin edges. This, a non maximum suppression has to be performed to thin out the edges. A non maximum suppression works by finding the pixel with the maximum value in an edge. Specifically, for a pixel q of the image, if its intensity is larger than the pixels in its gradient direction, then the pixel is kept as it is. Otherwise, the pixel would be set to 0 (turned black). This way, all the pixels that are not considered to be an edge are suppressed.

Step 5 Finally, hysteresis is used as a means of eliminating streaking. Streaking is the breaking up of an edge contour caused by the operator output fluctuating above and below the threshold. If a single threshold T1 is applied to an image, and an edge has an average strength equal to T1, then due to noise, there will be instances where the edge dips below the threshold. Equally it will also extend above the threshold making an edge look like a dashed line. To avoid this, hysteresis uses two thresholds, a high and a low one. Any pixel in the image that has a value greater than the higher threshold is presumed to be an edge pixel, and is marked as such immediately. Then, any pixels that are connected to this edge pixel and that have a value greater than the lower threshold are also selected as edge pixels.

Once all the steps are implemented the result is black and white image, where all the image's edges are colored white and everything else is black. The Canny detector is very fundamental for computer vision, because through the output image of the detector information about object of specific shapes or features. An example of the appliance of the Canny edge detector on an image can be seen on Figure 2.10.

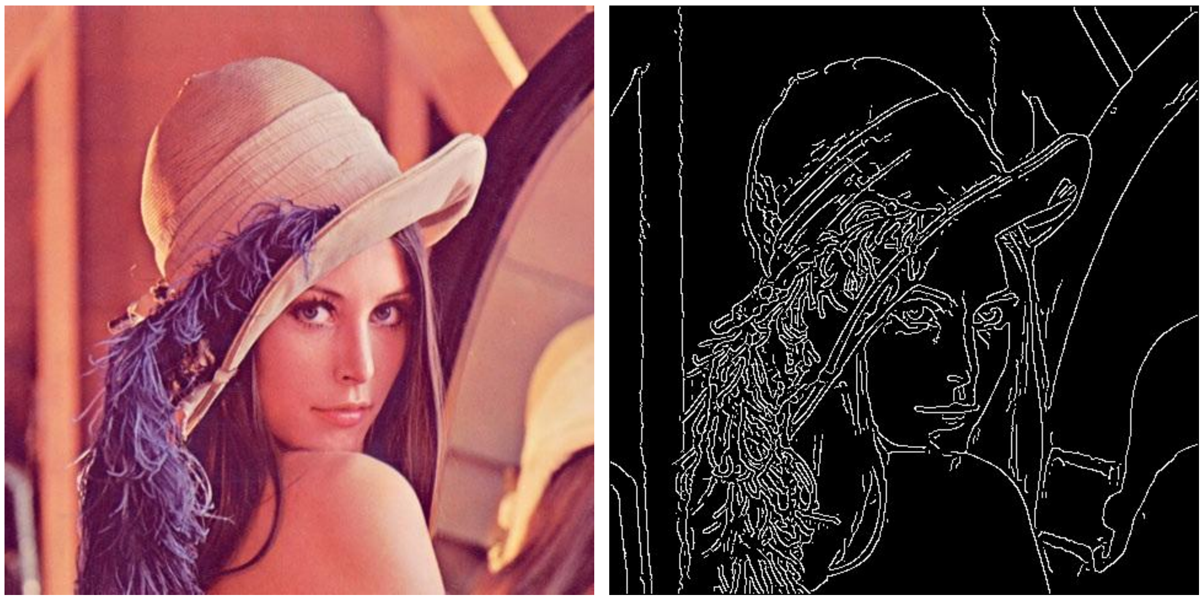


Figure 2.10: Source image (left) and Canny's output image (right).

2.6.2 Hough Transform

Hough Transform [31] is another one of the most common feature extracting techniques used in image analysis, computer vision and digital image processing. Its main purpose is to find imperfect instances of objects of a certain class of shapes, but the transform requires that the desired shape to be specified in some parametric form. This means that the Hough transform has many different forms. The classical Hough transform, though, that was proposed by Hough in 1962 was focusing on detecting lines in the image, but some of the other most commonly detected shapes through Hough Transform are circles and eclipses. The way this is achieved is by first determining specific values of parameters that characterize the desired pattern. Spatially extended patterns are transformed so that they produce spatially compact features in a space of possible parameter values. This way, the Hough transform converts a difficult global detection problem in image space into a more easily solved local peak detection problem in a parameter space.

The key ideas of the method [34] can be illustrated by considering a point (x_i, y_i) and the general equation of a straight line in slope-intercept form, $y_i = \alpha x_i + b$. Infinitely many lines pass through (x_i, y_i), but they all satisfy the aforementioned equation for different values of α and b. However, writing this equation as $b = -x_i\alpha + y_i$ and considering the αb-plane (also called parameter space) yields the equation of a single line for a fixed pair of (x_i, y_i). Furthermore, a second point (x_j, y_j) also has a line in parameter space associated with it, and this line intersects the line associated with (x_i, y_i) at (α', b'), where α' is the slope and b' the intercept of the line containing both points (x_i, y_i) and (x_j, y_j) in the xy-plane. In fact, all points contained on this line have lines in parameter space that intersect at (α', b'). These concepts are also called backprojection of the image point and can be seen in Figure 2.11, where the left figure shows a typical point image in the xy-plane, while the right figure shows the parameter lines produce by backprojecting image points into parameter space (c,m).

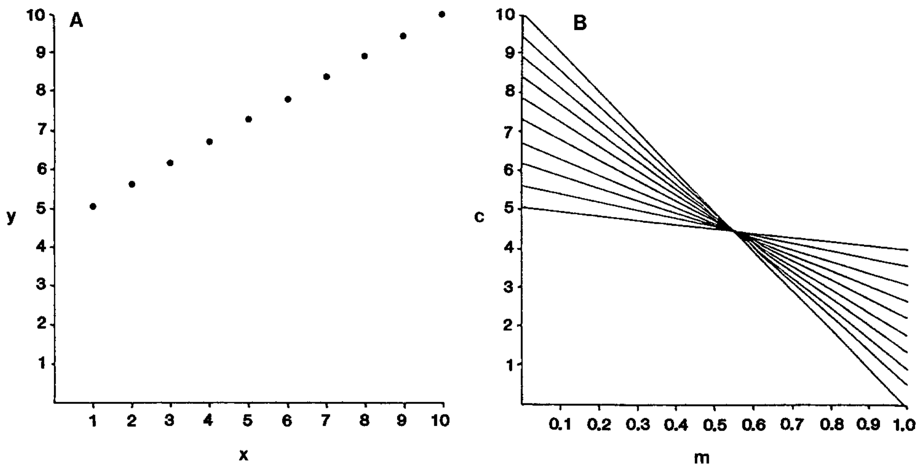


Figure 2.11: A backprojection example

The computational attractiveness of the Hough transform comes from subdividing the parameter space into so-called accumulator cells. Each of these cells (i, j) have an accumulator value of $A(i, j)$ and correspond to the square associated with parameter space coordinates (α_i, b_i). Initially, these cells are set to zero. Then, for every point (x_k, y_k) in the image plane, the parameter α equals each of the allowed subdivision values on the α-axis (of the parameter space) and solve for the corresponding b using the equation $b = -x_k\alpha + y_k$. The b's that are calculated this way, are then rounded off to the nearest allowed value in the b-axis (of the parameter space). If a choice of α_p results in solution b_q, then $A(p,q) = A(p,q) + 1$. At then end of the procedure, a value of Q in $A(i,j)$ corresponds to Q points in the xy-plane lying on the line $y = \alpha_i x + b_j$. The number of subdivisions in the αb-plane determines the accuracy of the colinearity of these points.

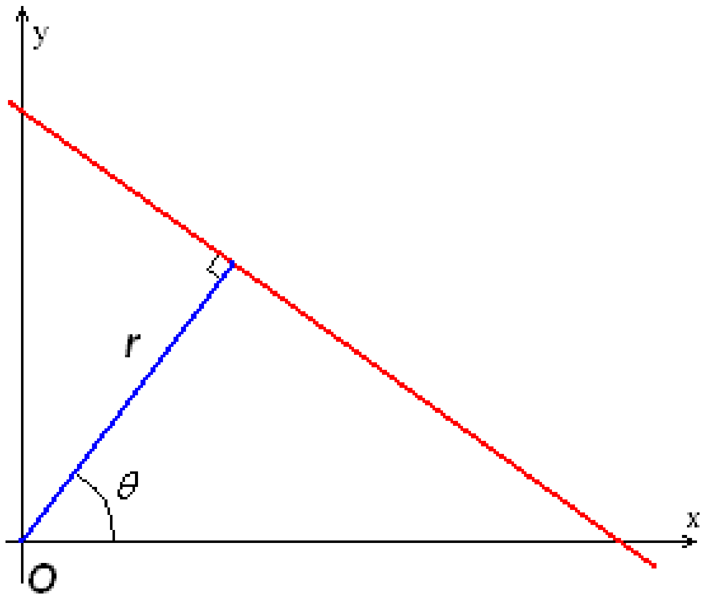


Figure 2.12: Parametrization of r and θ.[8](source)

However, using equation $y = \alpha x + b$ to represent a line comes with a problem. This problem is that the slope approaches infinity as the line approaches the vertical. One way around this is to use Hesse normal form are Duda and Hart proposed:

$$x \cos\theta + y \sin\theta = r\,, \tag{2.15}$$

where r is the distance from the origin of the closest point on the straight line, and θ is the angle between the x axis and the line connecting the origin with the closest point as shown in Figure 2.12.

[8] https://en.wikipedia.org/wiki/Hough_transform#/media/File:R_theta_line.GIF

The use of this representation in constructing a table of accumulators is identical to the method discussed beforehand. Instead of straight lines, however, the loci are sinusoidal curves in the $r\theta$-plane. As before, Q collinear points lying on a line $x \cos \theta_j + y \sin \theta_j = r_i$ yield Q sinusoidal curves that intersect at (r_i, θ_j) in the parameter space. Increasing θ and solving for the corresponding r gives Q entries in the accumulator $A(i, j)$ associated with the cell determined by (r_i, θ_j). The range of angle θ is $\pm 90°$, measures with respect to the x-axis. Thus, a horizontal live has $\theta = 0°$, with r being equal to the positive x-intercept, while a vertical line has $\theta = 90°$, with r being equal to the positive y-intercept or $\theta = -90°$, with r equal to the negative y-intercept.

2.6.3 Circle Hough Transform

Although the focus so far has been on straight lines, the Hough transform can be used to detect any feature or shape that can be modeled as a function of the form $g(v, c) = 0$, where v is a vector of coordinates and c is a vector of coefficients. The way that circles are being detected compared to straight lines is not very different and it is very straightforward. Specifically, the steps that have to be followed to detect circles in an image using the Hough transform are [31]:

- First, the accumulator cells have to be created, one cell for each pixel of the image. Initially, each cells value is set to 0 as it was done in the case of the straight lines.

- For each edge point (i, j) in the image, increase the values of the cells that could be the center of a circle. A cell can be considered to be the center of a circle through the equation $(i - \alpha)^2 + (j - b)^2 = r^2$, which basically is the equation of a circle.

- For each possible value of α found in the previous step, all the possible values of b that satisfy the previous equation are calculated.

- Finally, the local maxima that can be located in the accumulator space, once the whole procedure is completed, are the cells that represent the circles that were detected in the image.

However, the procedure above requires the a priori knowledge of the radius of the circles that must be detected in the image. If the circle's radius in not known, then a three-dimensional accumulator space can be used, to search for circles with an arbitrary radius. Also, the Circle Hough Transform can detect circles that are partially outside of the accumulator space, as long as there is enough of the circle's area in still present withing the image and thus the accumulator space.

Chapter 3

Problem Statement

3.1 Tangible Geographic Information Systems (GIS)

Natural disasters have, unfortunately, become a very common sight lately, with many recent witnesses being forest fires that wiped out hundred of acres of forest area, destroyed households and properties and sometimes resulted in human casualties. Due to the fact that most of these hazards prove to be extremely life threatening, it is essential for search and rescue teams to be well prepared to face and resolve them as quickly as possible. To that end, a tangible GIS, on which any natural disaster scenarios, like wildfires and floods, could be visualized, looks like a very useful tool for teams to have.

Tangible GIS are like common GIS, but they have the advantage that they can be altered in real time, so that they represent the geographical part that needs to be studied at a given moment or simulate changes that may occur to the landscape, for example due to a landslide. However, there is no easy way to capture all the changes that can happen on the GIS and in turn visualize the different shapes it can take. This means, that our system has to be able to measure the height of every point of the GIS and create a virtual image of it at any given time.

3.2 Simulation of Natural Disaster Response

In addition, since our goal is to create a natural disaster scenario, the addition of robust and realistic simulations of each of these disasters (fire, flood, earthquakes) becomes a very important part. Having said that, for each disaster that is going to be simulated on the tangible GIS, a corresponding mathematical model has to implemented, which will not only result in showcasing the disaster's behavior, but it will also adopt to the altering state of the GIS in real time.

Furthermore, these mathematical models require specific information regarding special factors that influence each different disaster's behavior. Some of this information can be provided manually to our system through an external source. This solution, however, does not fit the real time aspect of the tangible GIS, so a way to obtain the required information through the state of the GIS in any given moment has to be embedded into the system.

Last, but not least, due to the fact that the system to be developed aims at assisting search and rescue teams, a way to assist them through the visual projection of the GIS on the tangible material has to be added to the implemented system. A function that could prove to be useful for search and rescue teams could be the calculation of minimum-cost paths to help them get to the hazard source (e.g. fire front) or get away from it safely (e.g evacuation plans). That way the teams will have a better idea about which path is the best to follow depending on the points of importance on the map/GIS. They will even be able to simulate unexpected events, for example obstacles created by rumbles.

This thesis will focus mostly on the simulation of firefighting scenarios, providing the visualization of the landscape and the fire propagation, in addition to the generation of appropriate paths to the points of interest. However, the end result will aim to provide a system that could be easily adopted and used for the simulation of other disasters too.

3.3 Related Work

3.3.1 UC Davis Augmented Reality Sandbox

The Augmented Reality (AR) Sandbox [47] is tangible GIS project that was developed by the UC Davis' W. M. Keck Center for Active Visualization in the Earth Sciences (KeckCAVES), together with the UC Davis Tahoe Environmental Research Center, Lawrence Hall of Science, and ECHO Lake Aquarium and Science Center [9].

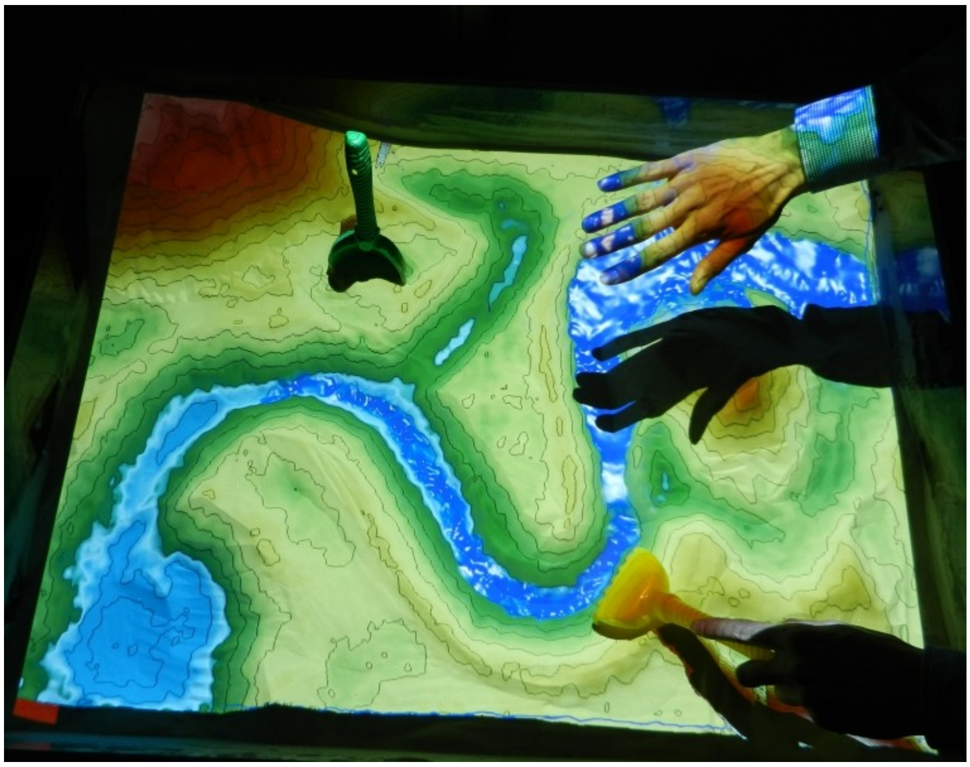


Figure 3.1: UC Davis' Augmented Reality Sandbox

The project combines 3D visualization applications with a hands-on sandbox exhibit to teach earth science concepts. The augmented reality sandbox allows users to create topography models by shaping real sand, which is then augmented in real time by an elevation color map and topographic contour lines. The users can also choose

[9]https://arsandbox.ucdavis.edu/

where water appears on the map and observe its behavior being simulated on the augmented map. A snapshot of the AR sandbox can be seen in Figure 3.1. This system can be quite instrumental in teaching geographic, geologic, and hydrologic concepts such as how to read a topography map, the meaning of contour lines, watersheds, catchment areas, levees, etc.

The goal of this project was to develop a real-time integrated augmented reality system to physically create topography models, which are then scanned into a computer in real time via a Kinect Sensor, and used as background for a variety of graphics effects and simulations. The final product is self-contained and relatively simple to get installed. All that is needed is a box filled with sand, a Kinect Sensor, a projector unit and a computer to run the specially developed AR Sandbox software. Its simplicity makes it ideal for use as a hands-on exhibit in science museums or schools.

The AR Sandbox is being used worldwide aiming not only to educate in an entertaining way, but also to raise public awareness and increase understanding and stewardship of freshwater lake ecosystems and earth science processes using immersive three-dimensional (3-D) visualization of lake and watershed processes, supplemented by tabletop science activity stations.

3.3.2 Simtable

Simtable [11] is another tangible GIS project that provides digital sandtables and customized agent-based models to the wildland fire, emergency management, defense and urban security communities, as well as colleges and universities. Based in Santa Fe, NM, the Simtable company was founded by Stephen Guerin and now Simtable is one of the few projects that combines agent-based modeling, data visualization and human computer interaction. Simtable provides a straightforward and friendly approach to incident response and training, by combining existing GIS data with next generation agent-based modeling and ambient computing.

Simtable features many functions, like agent-based modeling frameworks for exploring and visualizing complex scenarios involving physical and social phenomena. Also, its simulations can be coupled with mobile phones and tablets across disparate networks, allowing data to be shared from any location and at any time. Lastly, it allows the integration of any GIS for use in simulation and planning applications.

The whole application of Simtable can be controlled through a single laser pointer and allows the user to browse through a vast amount of parameters that he/she can change, in order to accommodate his/her own needs. All the information that the user needs is projected on the box filled with sand and the digital image of the Simtable is acquired by an integrated machine vision software installed on a camera. An instance of a demonstration of the Simtable to a firefighting team can be seen in Figure 3.2.

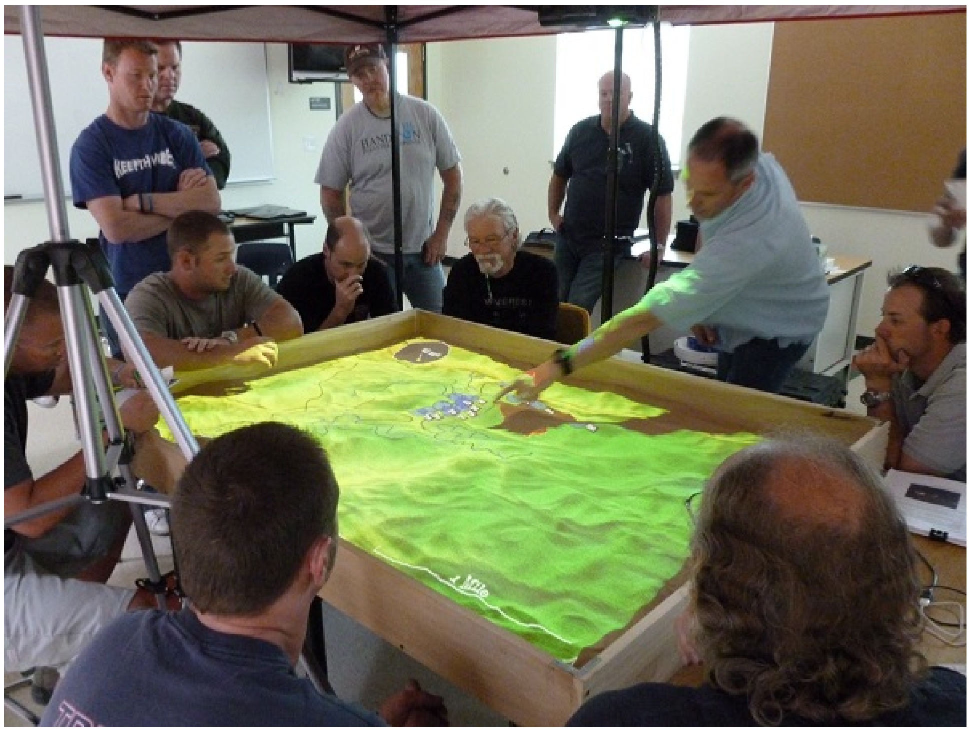


Figure 3.2: A Simtable demonstration

Chapter 4

Our Approach

4.1 Tangible GIS and Kinect Setup

First off, to solve our problem we need to start from the basics: the tangible Geographic Information System (GIS). In our approach we used a tangible GIS made out of sand (Sandbox/SandMap) very similar to the AR Sandbox [47] and the Simtable [11]. So, our GIS is basically a wooden box filled with white sand, which can be freely reformed according to the surface of the landscape that needs to be tested or studied. Also, the use of sand provides an easy way to simulate the appearance of unexpected obstacles and the occurrence of changes to the regular shape of the studied surface in real time. The sandbox that was particularly used in this thesis is shown in Figure 4.1.

A box filled with sand though is not enough to fully operate the tangible GIS. The user has to have some sort of means to observe and see whether the current form of the sand is the desired one. A way to do that is to project a visualized form of the surface on top of the sand. This visualized form can be either a display of different colors based on the elevation of the sand or even a projection of realistic graphics representing how the landscape might actually look like. This whole procedure can be done using a single projector unit placed above the sandbox and facing down towards it.

To be able to do that, all the information needed regarding the GIS has to digitized first, which in turn means that a sensor or camera of some sort has to be used. In our approach, a Kinect for Xbox One sensor was used and specifically the depth camera

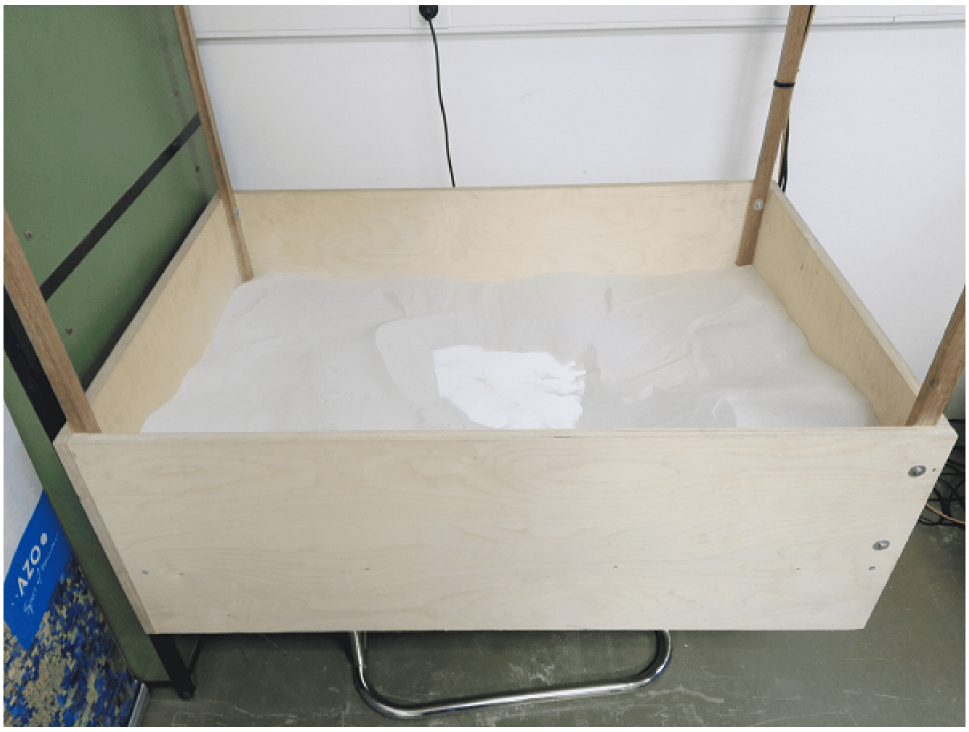


Figure 4.1: The Sandbox (SandMap)

part of it. The Kinect sensor is placed above the sandbox - right next to the projector unit - and facing down towards it, as shown in Figure 4.2. That way, the height of each point of the sand in the sandbox can be calculated through the depth camera of the sensor and can then be used to create a virtual image of the landscape. However, the Kinect sensor has to be placed high enough, so that the depth image output includes the whole surface of the sandbox that comprises the tangible GIS.

As for how to acquire the sensor's feed through a computer, all that is needed is the Kinect V2 SDK for Windows. The SDK provides various applications that showcase the capabilities of the Kinect sensor as well as code samples on how to obtain or use the sensor's feed. In addition, since our project was developed in Unity3D, the Kinect plugin for it had to be installed, in order to be able to use the Kinect's feed on Unity3D's projects and scripts.

4.2 Kinect Sensor Calibration

As mentioned in Section 2.1, the Kinect sensor for Xbox One consists of two cameras, one color and one time-of-flight, which serves as the depth camera. In systems that use cameras, the calibration of these cameras is a critical procedure for them to function properly. The calibration of a camera is the process of finding the true parameters that describe the camera system and can be characterized as either internal or external. The purpose of the internal calibration is to provide lens distortion correction and estimation of the camera's geometry, whereas external calibration estimates the position of the camera in relation to the scene.

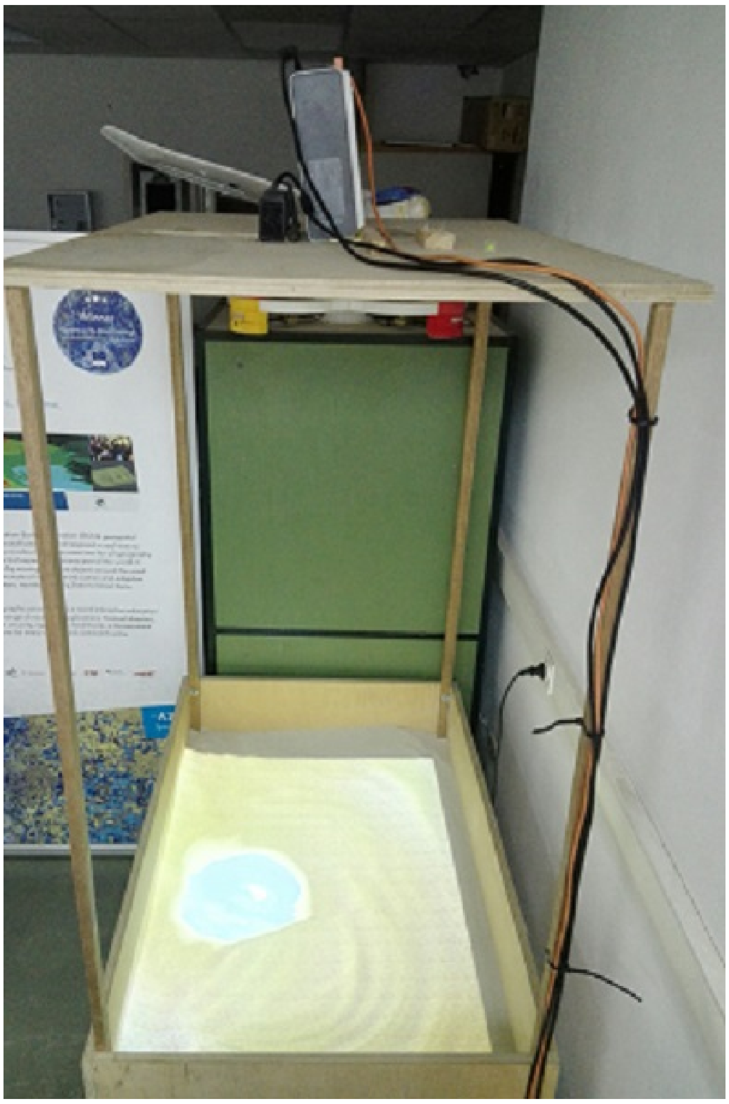


Figure 4.2: The Kinect and Projector Setup

Unfortunately, the Kinect sensor for Xbox One does not provide any means for manually calibrating its cameras. This is because the Kinect software installed on all Kinect sensors automatically calibrates them, in order to fit any environment they might be used in. However, this factory-installed internal calibration process is adequate for our purposes.

On the other hand, the Kinect SDK provides no means of external calibration. This means that positioning the Kinect sensor over the sandbox, as described above, may provide a depth image that will include parts of the sandbox or its surroundings that need not be studied. The Kinect SDK provides no way of narrowing the view of the depth camera to prevent that. Therefore, the implemented system will have to analyze unnecessary data or even proceed to the visualization of parts that will not give an accurate approximation of the desired landscape. This in turn means that the procedure of digitization will require more time and resources to complete. One way to deal with this problem is to reposition the Kinect Sensor at a height where the depth image will have no unwanted data. However, this solution could be impractical for usability reasons.

In our approach, a custom calibration solution was implemented. Specifically, using a Unity3D graphical application, the user is able to select only the part of the depth image that is necessary. In this application, the user can see the full view of the color camera of the Kinect sensor and select the area of interest. The reason behind using the color camera rather than the depth one is to provide a clear image of the GIS, since it can prove to be more difficult to find the exact edges of the sandbox on the depth image. The user can select any part of the sandbox (which basically is the tangible GIS) by drawing a box over the desired area. Note that, by construction, the Kinect Sensor's image frame is aligned with the sandbox frame. Figure 4.3 shows a snapshot of the external calibration procedure, where the selected area covers the entire surface of the sandbox.

However, the user chooses an area based on the color image of the Kinect sensor, whereas the information needed to form a 3D image of the surface of the sand comes from the depth image. This wouldn't be an issue if both the color and depth image had the same resolution, but as mentioned in Section 2.1 the depth image is of lower resolution than the color one. That makes a mapping between the two images vital for the system to work properly. A mapping procedure would take a point on the sandbox and its coordinates on the color image and calculate the exact same point's coordinates on the depth image.

This mapping procedure had to be embedded onto the calibration application. To realize this we had to extract two diagonal end points of the box the user drew on the sandbox image and map them onto the depth image. The color camera's image texture

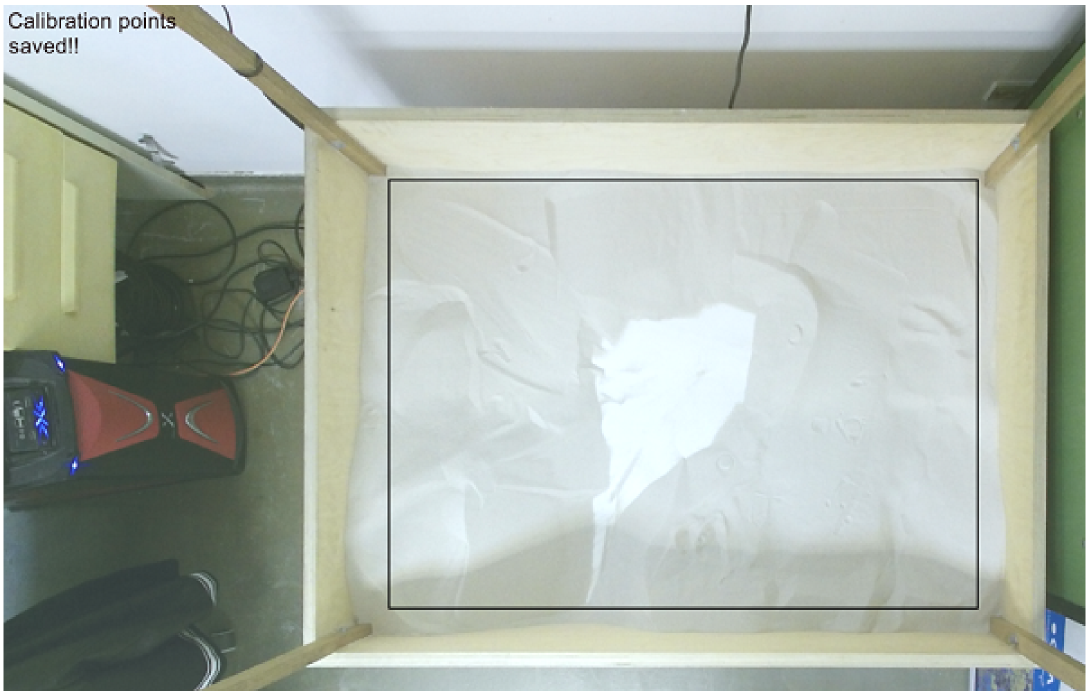


Figure 4.3: Snapshot from the Calibration Procedure

is attached to a Unity3D renderer. The end points of the box are captured through this renderer, while the user draws the box. Once the points are extracted, the Kinect SDK and Unity3D plugin provide a mapping function between the color and depth image of the sensor, which takes as input a point in the color image and returns the same point on the depth image. Using this function the desired points on the depth image can be stored, in order to be used whenever necessary.

4.3 Depth Data Extraction and Visualization

Visualization is the process of viewing and manipulating a real life 3D object through a computer, by using some sort of visualization software. This process replaces all the drawing procedures that had to be done by hand and so it can be very helpful in simulations of different natural phenomena, 3D-modeling and rapid prototyping. There are many visualization tools that can provide a 3D-model of basically anything that needs to be studied or simulated. The Kinect SDK provides a similar tool that allows the user to extract a 3D model of its depth image and therefore of all the objects that are placed in front of its camera. However, this particular tool comes with the same problem addressed during the calibration process where unnecessary data were also included in the image.

The Unity3D game engine is a very good 3D-modeling and visualization tool and works very well with the Kinect sensor, because of the plugin provided for it. Also, the Kinect's depth camera view can be calibrated through Unity3D - as mentioned in the previous section - so that the output model of the sand does not contain any excess data. In this thesis, a Unity3D mesh was used to implement the 3D model of the sand's surface, because of its flexibility in size and style.

4.3.1 Kinect Depth Data

The Kinect sensor's data can be deciphered into two types: depth and color, as shown in 4.4. Both of these data types can be obtained in forms of arrays. These arrays are basically a digital representation of the actual images of the Kinect sensor and contain values for each pixel of both images. Although one could easily assume that the obtained arrays are two-dimensional, since images of various types can be defined by two-dimensional arrays, this is not exactly the case. The depth image data is stored in a one-dimensional array of `integer` type and the color image data is stored in a one-dimensional `byte` array. In the case of the color image there is additional information that can be obtained for its texture, which will prove useful later in this thesis.

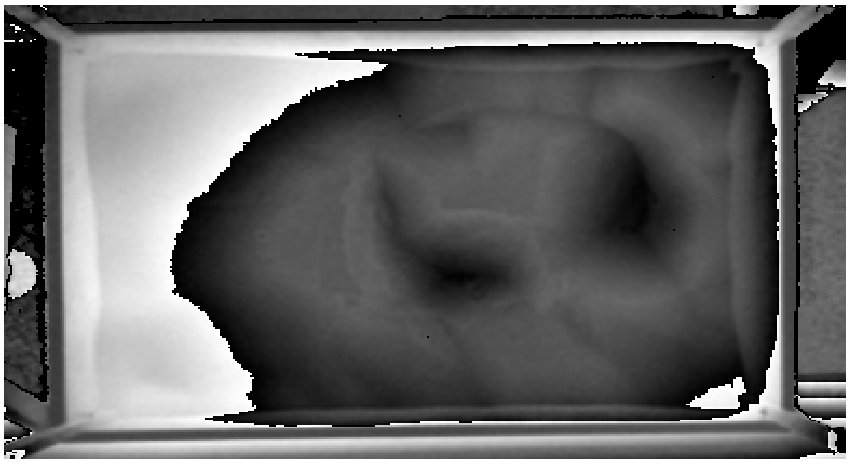


Figure 4.4: A snapshot of the Kinect's Depth Image

The extraction of these data can be done through the Kinect sensor's plugin by acquiring its frames. In order to do that, there is a specific sequence of variable assignments that needs to be implemented. The Kinect plugin provides two classes, one for the sensor and one for the frame reader, that can be used to obtain all the necessary data. So, first of all, the Kinect sensor has to be assigned to a variable of the sensor class type. If there are more that one Kinect sensors connected to the computer at the time of the assignment, then the user has to declare from which of the sensors he wants to get frames from. Otherwise, the sole sensor connected is assigned to its corresponding variable. Every sensor variable contains a reader which can easily be assigned to a variable of the frame reader type. However, the reader's type has to be defined first. There are three distinct reader types, a color frame reader, a depth frame reader and a multi source frame reader (both color and depth). The multi source frame reader was the one picked in this thesis.

Once the two core variables (the sensor and the reader) are declared, the frames of the sensor can be obtained and stored to their respective variables. Since, in this thesis, the multi source reader is used, the frames that are acquired contain both color and depth data, which can be extracted and stored to their corresponding arrays and variables (when it comes to texture data). The frames also contain information about the dimensions (width and height) of the images which can be used to convert the two one-dimensional arrays into two-dimensional arrays and define the size of the mesh that has to be implemented.

However, the information that is stored in these two arrays contains all the excess data that don't need to be studied. In order to get rid of the unnecessary depth data, the array that contains them has to be "cropped". The two points defined during the calibration procedure are basically (x, y) coordinates that correspond to actual points of the depth image and can define a new smaller depth image. The same cropping procedure takes place also over the color image of the sensor. Figure 4.5 shows both the original images and their corresponding cropped versions of the color and the depth images.

Once the new depth data array is created, the next frame is acquired and the whole procedure is repeated for every next frame. However, this doesn't have to be done for every single frame that is acquired from the sensor's reader, since most of the time there won't be any change made onto the tangible GIS. This can easily be avoided by

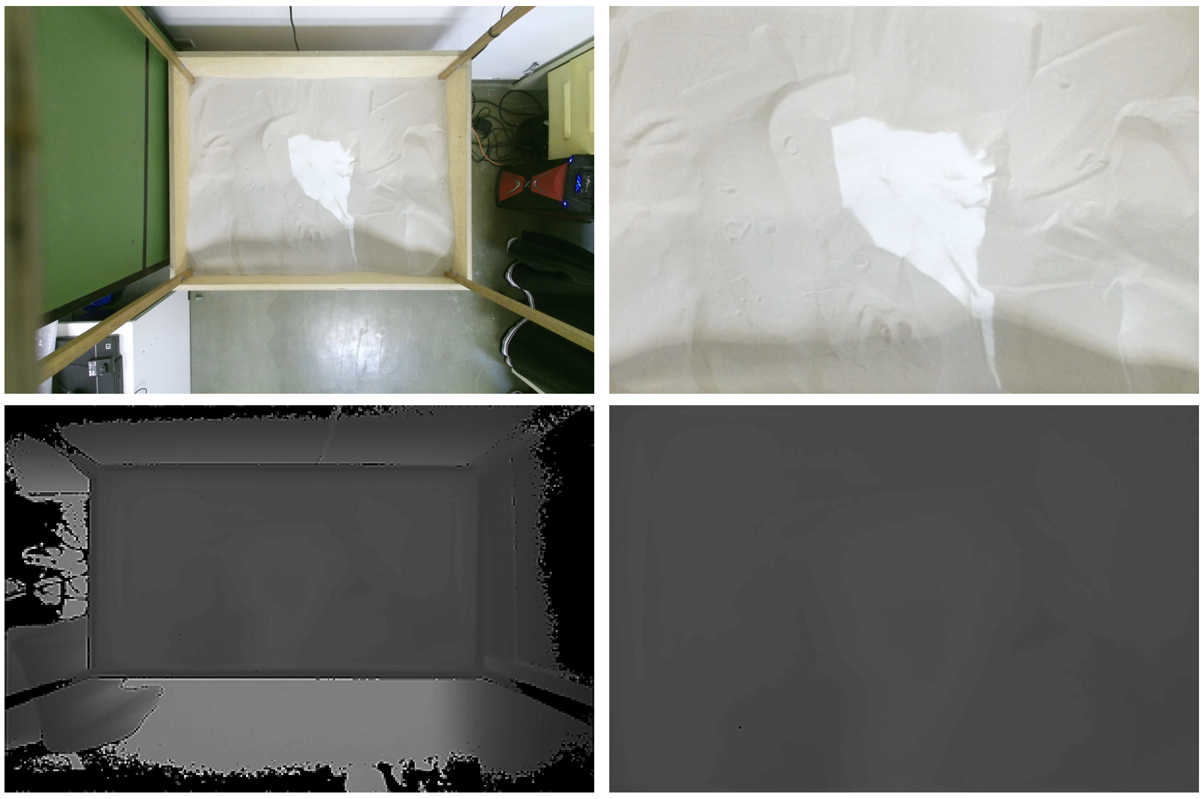


Figure 4.5: Full (left) and cropped (right) color and depth images of the Kinect Sensor

comparing the current depth frame's data with the data that are stored in the array, in order to see if there were any changes made to the map. This, though, proves to not be a very easy task, because the Kinect's depth image contains noise, which can be interpreted as small changes to the map. A smart way to avoid this is by checking for changes and replace the stored data only if a non-negligible percentage (approximately 2%) of it is different than the current one.

4.3.2 Hand Interference Handling

Changes on the sand map are mainly caused by human intervention or basically humans using their hands. This approach focuses more on hands, because the tangible GIS is made out of sand and the main way for someone to change its shape is by using his/her hands. However, there are two ways hands can intervene in the depth image of the sand map. One is by actually messing with the sand's shape and the other is by just placing the hands above it. This calls for a way to recognize each of these scenarios and act differently in each case.

The way to do this is relatively easy. Once a hand is placed above or touches the sand it immediately becomes part of the depth image that is obtained through the Kinect sensor. This means that the depth data array will contain different depth values for the points that correspond to the position of the hand. These values, though, will not be normal compared to the depth values that where stored before the hand was inserted in the scene. To define a normal depth value, the maximum and minimum depth of the sand has to be defined first. This is an easy task, since all the depth values (before the hand is placed) are stored in the depth data array, so all that needs to be done is find the maximum and minimum values among those. Once these are found, normal depth values are considered those between the two extreme values.

So, in the case where a hand is placed above the sand, the points of the array that correspond to the position of the hand will contain values that are much lower than the minimum depth value (the minimum depth value corresponds to the closest point to the sensor). When this happens, the hand has to be ignored, considering that if the hand is higher than the sand's surface, no changes are being made to the sand. However, when a hand is used to change the shape of the sand, the depth values corresponding to it will be lower than the minimum value but not so much as to be considered abnormal. In this case, the depth data that are stored have to be changed, in order for the changes to be displayed on the sand in real time. Once the hand gets out of the sandbox or is raised higher above the sand then the depth data stops changing and thus the data in the depth array stops being replaced by the current data, as they did before the hand was inserted in the scene.

Although not used in the thesis, the placement of hands above the sand could be easily recognized and used as input for other functions.

4.3.3 Depth Data Visualization

4.3.3.1 Conversion of Depth Data to Normalized Height Data

As briefly mentioned above the depth data that are collected from the Kinect sensor contain noise, that is caused internally. This noise appears in the form of zero or very high ("spikes") values and can create problems during the definition of the polygon mesh's vertices. A solution for this problem can be the normalizing of all the depth data values, so that they range from zero to one according to how close they are to their minimum and maximum value respectively.

Since, the minimum and maximum depth values have already been calculated at the hand interference handling step, the normalization procedure can be easily implemented. However, the whole process of the visualization of the tangible GIS could be made much easier if all the obtained data where referring to the height of the points of the sand rather than the depth. This is because calculating the height instead of the depth will give a more clear image of the 3D object that needs to be created and will better match the 3D vertices of the Unity3D's modeling tool.

In this approach, this conversion is an easy task, because of the Kinect and sandbox setup that was implemented. Specifically, the Kinect sensor is placed above the sandbox and facing down on it. This means that an assumption can be made, that the negative values of the depth data can correspond to the height values of the same points and in turn the maximum and minimum depth values can be converted to the minimum and maximum height values respectively. So, instead of the depth values being directly normalized, they are first converted to height values and then these heights values are scaled at a range between 0 and 1, according to their proximity to their respective minimum and maximum height values.

This procedure will return a noise free map of positive values that can be directly used to define the shape of the mesh through its vertices. Also, since the values are all normalized they can easily be modified by using special multipliers according to where they need to be applied.

4.3.3.2 Mesh Generation

Once the height data are calculated, all the data that are needed for the generation of a polygon mesh in Unity3D are fully defined. To generate the polygon mesh in Unity3D, an approach very similar to the approach of Lague [10] for creating a procedural landmass was followed. At first, the data that are essential to create a mesh are its vertices and its triangles (since meshes in Unity3D are triangular). To define the vertices all that needs to be done is to associate the height data to x, y, z coordinates, that can then be used to declare the mesh's vertices. This is fairly simple, because the height data after the normalization procedure are stored in a two-dimensional array of a specific width x and height y. This means that the x, y coordinates of the height data array can be translated and used as the x, y coordinates of the mesh's vertices, while the height value itself can be the z coordinate of its corresponding vertex. However,a small adjustment has to be made. In Unity3D the two coordinates that refer to the 2D aspect of a 3D object are x and z, while y is referring to height or depth. This means that the x and y coordinates of the height data array have to be translated onto the x and z coordinates of each mesh's vertex and the height value onto the y coordinate of it.

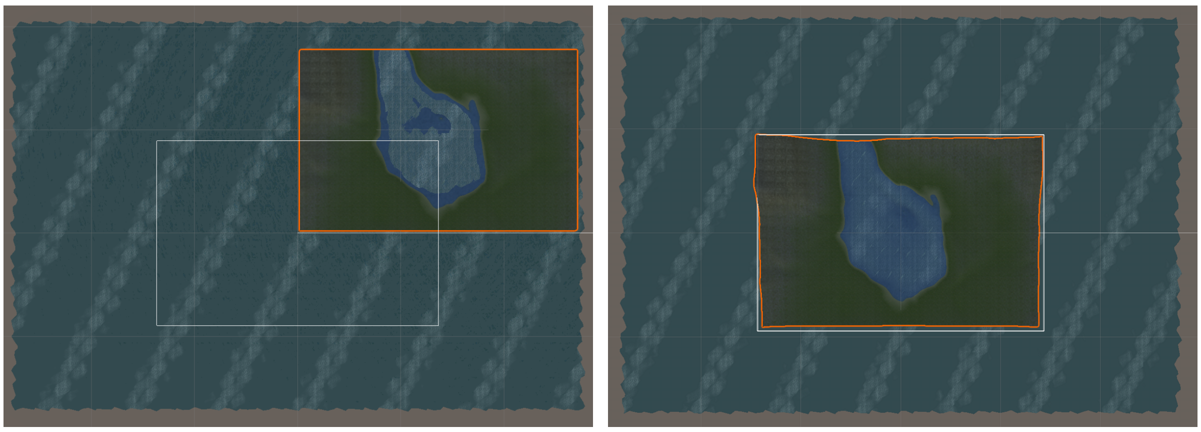


Figure 4.6: Not centered (left) and centered (right) Mesh

Defining the vertices this way though can lead to a problem where the final mesh is not positioned right at the center of the scene and in turn lead to adjustments to the game camera of Unity3D in order for the top down view to include the whole mesh. To prevent all these, a small change has to be made so that the mesh is placed directly at the center of the scene. The center of the scene is the point where all x, y, z coordinates are zero. The height data values' x coordinate ranges from zero to the width of the depth image and the y coordinate from zero to the height of the image. This means that in order to center the mesh, the middle value of the height data array, whose x, y coordinates are equal to $(width/2, height/2)$, has to be translated into the centered vertex of the mesh. This can be done by shifting all the vertices left and up (into the negative x and z values) by $width/2$ and $height/2$ respectively. In Figure 4.6 the mesh before and after the centering procedure can be seen.

After all the mesh vertices are calculated, its triangles have to be defined as well. A triangle is basically a connection between three of the previously defined and stored vertices. The correlation between those two happens through two one-dimensional arrays, as mentioned in Section 2.2.3, one being the one where vertices are stored and another where indices to vertices are stored, defining the three vertices each triangle is comprised of. However, a triangle can not be created by three vertices that are all on the same row or column of the vertices map, but a single vertex can define two triangles, as shown on Figure 4.7, meaning that when defining the mesh's triangles, the bottom ($y == height$) and far right ($x == width$) vertices do not define any triangle.

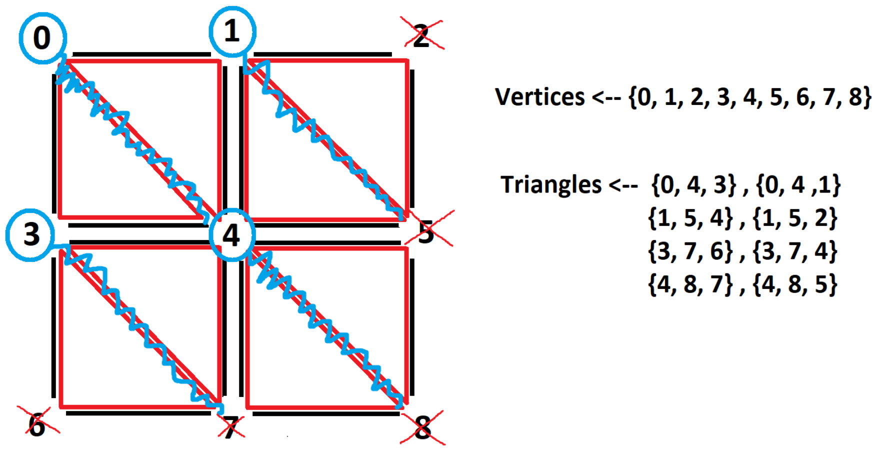

0
1
2
3
4
5
6
7
8
Vertices <-- {0, 1, 2, 3, 4, 5, 6, 7, 8}
Triangles <-- {0, 4, 3} , {0, 4 ,1}
{1, 5, 4} , {1, 5, 2}
{3, 7, 6} , {3, 7, 4}
{4, 8, 7} , {4, 8, 5}

Figure 4.7: Triangles definition through vertices (Crossed out vertices don't define any triangles)

Now all that is left, is the creation of a mesh `Gameobject` in Unity3D. A mesh `Gameobject` consists of two basic parts: a mesh filter and a mesh renderer. The renderer part is responsible for how the mesh will interact with the lighting of the scene and for its texture, which will be addressed later in this thesis. In this phase of the project, the filter of the mesh is the part where the vertices and triangles, that where estimated before, have to be applied. This application on the filter will result in the creation of a mesh, the shape of which will be the current shape of the tangible GIS. Since the data that are being translated into this mesh are dynamically changing, so is its shape.

4.3.3.3 Mesh Textures

At this point, the part that is missing, in order for the mesh to be a realistic representation of the desired landscape that has been shaped on our tangible GIS, is coloring its triangles according to what each one of them is considered to be depicting.

As mentioned in Section 2.2.3, this can be done through Unity3D's texture materials, but first the UV coordinates for each vertex have to be defined. The purpose of the UV coordinates can be simplified to "telling" each vertex where it is in relation to the rest of the map as a percentage of both the x and y axis. So, the UV coordinates array is basically a `Vector2` array of the same length as the vertices array that stores percentages calculated by dividing each x and y value of the height data array (which are also translated to the vertices x and z coordinates) with the array's width and height respectively.

Once the UV coordinates are defined, all that needs to be done is apply the texture to the mesh. In Unity3D, a texture is applied to a 3D object through materials. Materials are like real life paints that can be modified in terms of color, smoothness and transparency and can be applied to 3D objects in order to enrich them with more details. Since, the generated mesh is considered a 3D object on its own, materials can be applied on it. However, in this case, coloring the mesh is not as simple as just applying a material on it, because different parts of the mesh have to have different textures according to which real landscape area they correspond to. These distinct areas - and thous textures - are sea, sand, grass/plain fields,

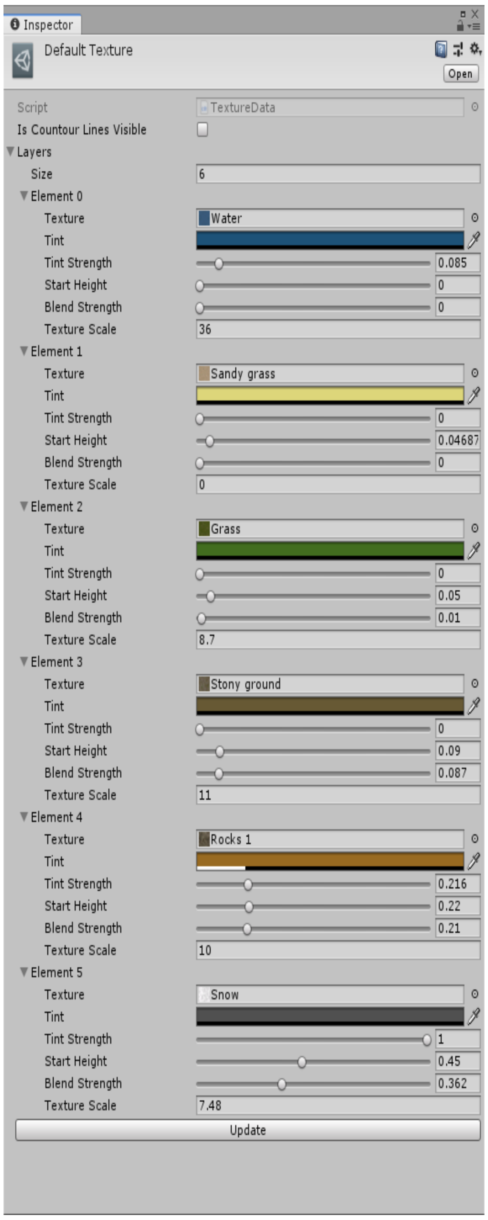

```
Inspector
Default Texture                              Open
Script                      TextureData
Is Countour Lines Visible   ☐
▼Layers
  Size                      6
  ▼Element 0
    Texture                 Water
    Tint                    🟦
    Tint Strength           0.085
    Start Height            0
    Blend Strength          0
    Texture Scale           36
  ▼Element 1
    Texture                 Sandy grass
    Tint                    🟨
    Tint Strength           0
    Start Height            0.04687
    Blend Strength          0
    Texture Scale           0
  ▼Element 2
    Texture                 Grass
    Tint                    🟩
    Tint Strength           0
    Start Height            0.05
    Blend Strength          0.01
    Texture Scale           8.7
  ▼Element 3
    Texture                 Stony ground
    Tint                    🟫
    Tint Strength           0
    Start Height            0.09
    Blend Strength          0.087
    Texture Scale           11
  ▼Element 4
    Texture                 Rocks 1
    Tint                    🟧
    Tint Strength           0.216
    Start Height            0.22
    Blend Strength          0.21
    Texture Scale           10
  ▼Element 5
    Texture                 Snow
    Tint                    ⬛
    Tint Strength           1
    Start Height            0.45
    Blend Strength          0.362
    Texture Scale           7.48
                 Update
```

Figure 4.8: The texture layers used to color the Mesh

stony grounds, rocks and snow and the main factor that determines where each of these textures should be applied is the height data obtained through the Kinect sensor.

In order to do that, a custom shader is needed. Materials in Unity3D consist of two parts: a shader and its shader properties. The shader part is the one that is responsible for the texturing/coloring of the material according to the properties that are bestowed upon it and can be controlled through scripts. So, first, the properties of the shader have to be declared. This was implemented following the texture logic that was described above. Through this procedure, a texture data component was created in Unity3D, which contained , as shown in Figure 4.8, six layers of textures each corresponding to one of the six distinct landscape type and information about the height threshold that determined the range of values on which each texture should be applied, the percentage in which each texture should blend in with the other textures, their individual scale and how they should interact with the scene's lighting (tint).

These texture data were then applied as the properties of the custom shader , which was then programmed to color each part of the mesh according to these data/properties. This means that each layer of texture was applied to the part of the mesh whose height value was corresponding to the height threshold that was defined to it, so that the areas corresponding to sea would have the water texture applied, the grass areas the grass texture and so on. The use of the custom shader also has an advantage over the use of different distinct materials, because it can update the texture of the material it is applied on in real time. This makes it perfectly fitting to this thesis, since the shape of the mesh has to be the real time representation of the tangible GIS's surface, meaning that the colors and textures on it have to be updatable in real time as well for the whole procedure to be robust.

4.3.3.4 Mesh Smoothing

The shape of the mesh that is generated through the procedure described above bears a very close resemblance to the actual shape of the tangible GIS. However, a few problems appeared when it comes to its visual accuracy. One of these problems was that the height of the mesh was not enough to clearly display the 3D surface of the map. That problem was am excepted one since the normalized height values all ranged for zero to one, which although gives an accurate representation of the sand's surface, the 3D object that was created was relatively small and thus some key parts of the sand's shape where not as clear as they should be. Thankfully, normalized values can be easily modified by using multipliers, which is what was implemented in this approach. Specifically, during the definition of the mesh's vertices all height values where multiplied by a height modulator, which could be modified according to the needs of each appliance. This way, the height differences between for example the land and the sea were clarified and could be spotted clearly.

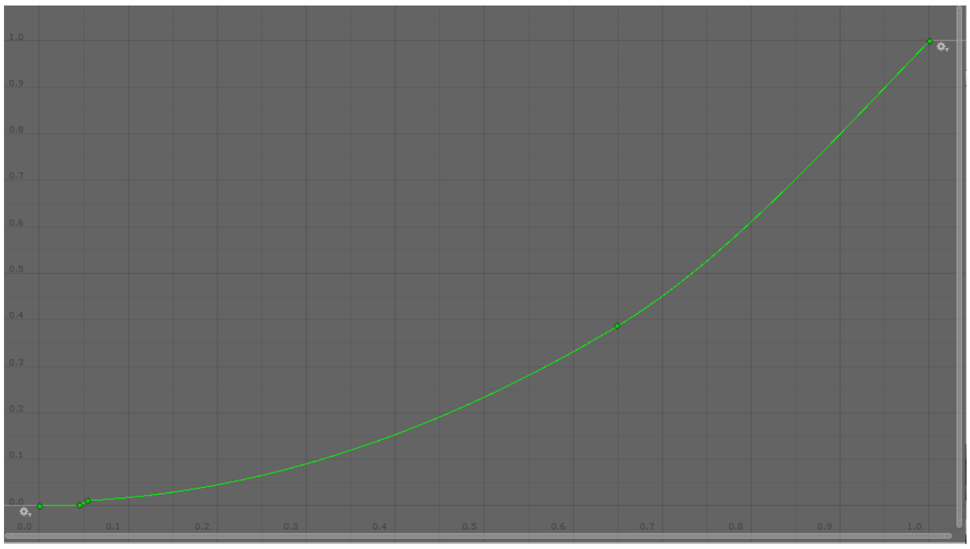


Figure 4.9: The animation curve used to smooth the Mesh

Another problem was that the surface of the mesh was not as accurate as it should be in some parts of it. These parts where mainly the parts where sea should be displayed. In particular, the surface of the parts that were considered to be sea appeared to have small hills on them, while the accurate representation would be of a smooth surface. This problem was not caused due to noise or bad implementation, but it was an inevitable occasion, where data of different height values were identified as sea surface, because they were below the threshold that was set on the previous step. This can easily be dealt with by using an Animation Curve. An Animation Curve is, as its name suggests, a curve that can be used in Unity3D for adjusting values associated with either animations of gameobjects or generally graphics. Its purpose is to give a more smooth movement to the animation that it is attached to or even to force a specific animation according to some stored values dictating how these animations should look like. The animation curve that was used in this thesis is shown in Figure 4.9, while the results of applying the animation curve on the mesh can be seen in Figure 4.10.

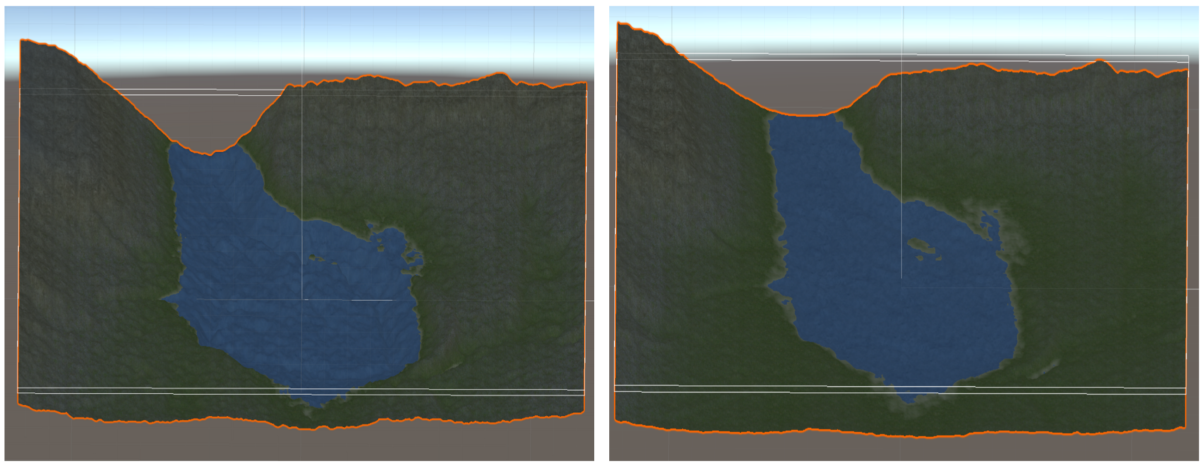


Figure 4.10: The Mesh before (left) and after (right) the use of the Animation Curve

An animation curve, however, can also be used on meshes to adjust the values passed onto its vertices, in order to give the mesh a smoother and curvy like shape. This is done by defining the shape of the curve itself according to how the values applied to the mesh have to be adjusted, so that the final shape of it will be the desired one. This adjustments to the values are being done through the axis of the curve. The x-axis of the animation curve corresponds to the normalized values that need to be adjusted while the y-axis determine the value to which the initial value will be adjusted to. This means that as the shape of the curve changes so will the values to which it will be applied, in order to correspond to the new shape of the curve. In this approach, an animation curve is used to adjust the normalized height values that are assigned to the mesh's vertices, in order to give a more flat shape to our mesh at the parts that are considered to be sea and non-rocky land and a more mountain-like look to the areas where height values are higher and the surface should look like a rocky mountain.

Lastly, even after these all these smoothing procedures that have been implemented on the mesh's surface, there are still some small spikes that appear mostly on the mountain areas. This is more of a visual bug rather than a problem caused due to noise or abnormal height difference of the stored data. To fix this issue, the smoothing approach [12] was applied to this thesis. This approach is basically a repetitive appliance of smoothing filters, such as Laplacian and Humphrey's Classes, on the mesh data, in order to get rid of existing distortions of the mesh's surface. In our approach, only the Laplacian filter was applied on our mesh data and as experiments proved a ten times appliance of the filter was enough to smooth out the surface of the mesh itself. The smoothed mesh is shown in Figure 4.11.

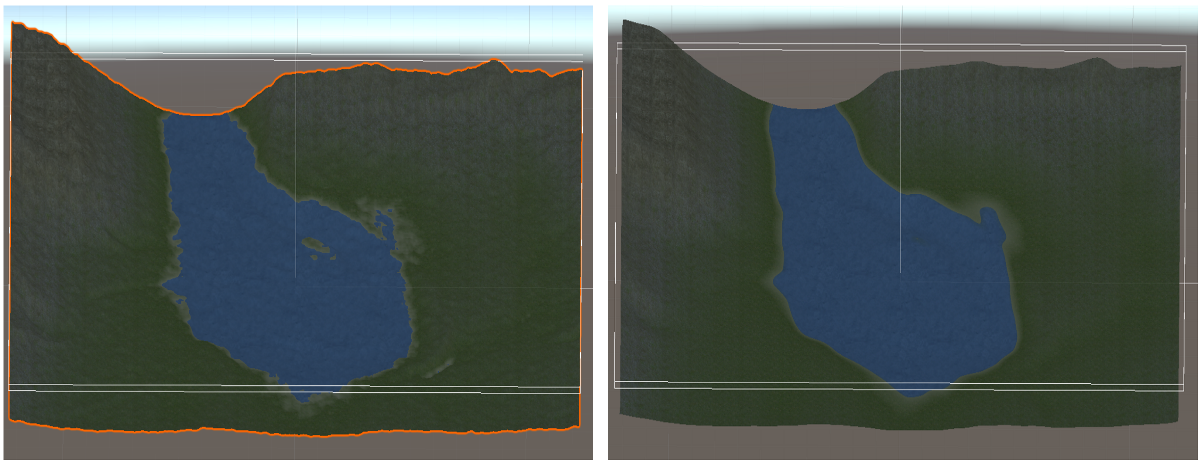


Figure 4.11: The Mesh before (left) and after (right) the appliance of the Laplacian filter

Once all of the procedures described above are implemented, the output that is created is a noise and surface distortions free mesh, the shape of which is based on the depth data that are obtained through the Kinect sensor and then converted to height data, while the textures that are applied on its triangles are also determined by the same data. In other words, the final mesh that is generated constitutes a very realistic real time representation of the landscape that needs to be studied and that is shaped on the sand of the tangible GIS.

4.3.3.5 Camera Positioning and Water Addition

In Unity3D any application/game that is created need a camera gameobject in order for it to be complete. Each scene in Unity3D can have multiple cameras providing various views from different angles. These views are what most commonly is referred to as "player views", which means is the view that the user of the application or the game (thus the name "player") will have when the application is launched. In this approach, though only one camera needed to be used but it had to be positioned so that the final view would be a top down view of the mesh. This was done by simply switching the projection of the camera from perspective to orthographic, rotating it by 90° on the x-axis and finally moving it to the right position so that its view included the whole mesh.

The view of the final mesh that was created was a very realistic one and the mesh itself was a very good representation of the landscape that had to be modeled. However, some tweaks could be done to the whole scene to make it look even more realistic. One of these tweak was adding another element to the scene: water. Specifically, the idea is to add water particles at the parts of the mesh that are layered as water. This way, the projected model will not only look more realistic but it will be more visually appealing and playful looking among kids, making it more accessible as a learning tool.

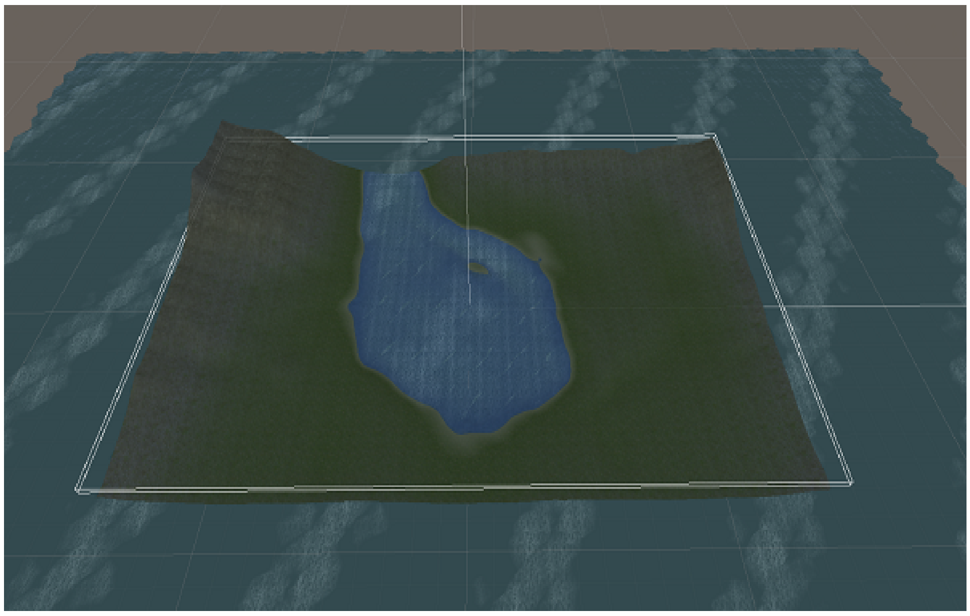


Figure 4.12: The final Mesh with the added Water prefab

To implement this, the environmental asset that is provided by Unity3D for water was used. In detail, the water asset that is available in the current Unity3D build, provides two options for the water particles, through the form of prefabs (prefabs will be discussed later in this thesis). The first option is "Basic" and the other one is "Advanced". The difference between those two lays on the textures that each of these prefabs has applied to it. Basically, the only difference is that the "Advanced" water option has more texture to make the water feel more realistic and natural.

In this thesis, the "Advanced" water option was used and the way that was applied to our already created scene, was by attaching the water prefab to an empty gameobject. The water gameobject consists of four tiles, which are specially created meshes that are programmed in such a way, so that they look and behave like real water would do. In our case, though, the dimensions of the water gameobject exceeded those of the mesh. However, this was not a problem, since the camera that was used for the top-down view of the mesh was positioned in such a way to only focus on the mesh. The only adjustment that had to be done to the water gameobject, was moving it higher (or lower) so that the water could be visible in the parts of the mesh that corresponded to it.

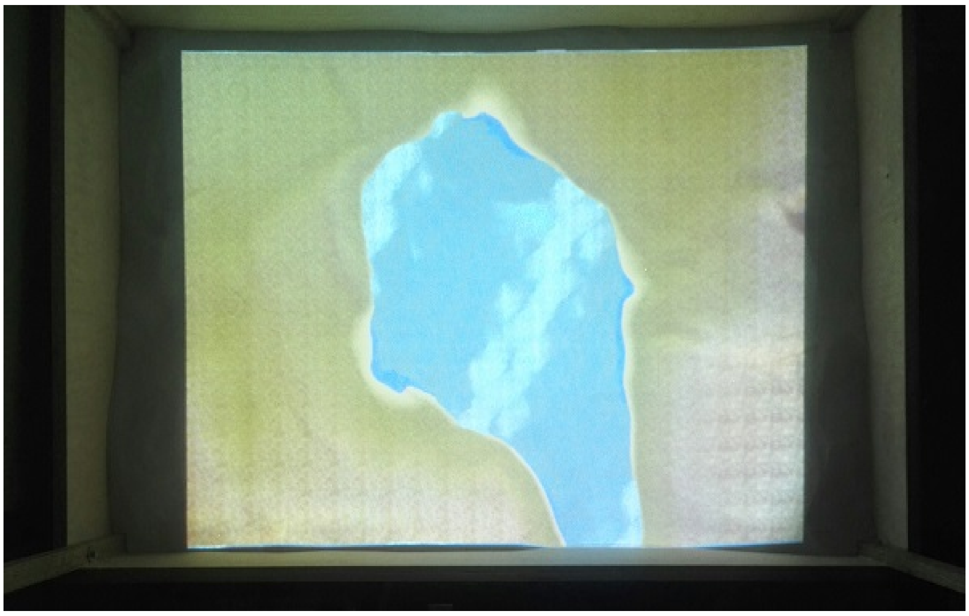


Figure 4.13: The final 3D model projected on the sand

So, after these tweaks were completely implemented, the basic part of our implementation was ready to be tested. In our approach, there were two types of tested that had to be carried out. The first type is a test in the editor of Unity3D, in order to ensure that everything works fine. The second type is a test on the sandbox, which basically will prove if the implementation is correct or if adjustments have to be made. The final image of both the 3D model of the GIS's surface and the projection of it on the sandbox are shown in Figure 4.12 and Figure 4.13.

4.4 Fire Propagation Simulation

4.4.1 Wildfire Spread Modeling

At this point of our project, the most basic element of simulating a firefighting scenario, the visualization of the landscape on which it is going to take place, has been fully implemented. So, the next most essential element to a firefighting scenario is the simulation of the fire propagation. In this approach, a cellular automaton based model was implemented to model the spread of the fire, due to the fact that it can be perfectly combined with geographic information systems, like the one in this thesis.

4.4.2 Cellular Automata Based Model

4.4.2.1 Grid and Cells' Parameters Definition

Specifically, in this thesis, the model that was implemented was based on the work of Mr. A. Alexandridis, Mr. D. Vakalis ,Mr. C.I. Siettos and Mr. G.V. Bafas on creating a cellular automata model for forest fire spread prediction for the case of the wildfire that swept the island of Spetses in Greece in 1990 [16]. However, a few modifications had to be made on their approach, because all the factors about the vegetation of the land and the wind's speed and direction (as well as the wind's behavior throughout the fire event) are known beforehand, since the study is based on an event that has already happened. In our case, all the factors that affect the fire's behavior must be estimated in real-time as the surface of the tangible GIS can change in any given moment.

First of all, the basis of a cellular automaton is its grid (fire-grid), the cells of which are going to contain all of the information needed for the fire propagation simulation. The size of this grid depends on the size of the map that needs to be studied. In our case, this map is the surface of our tangible GIS, which means that the size of the grid must be equal to the size of the array where the depth/height data are stored. So, basically, the fire-grid is a two-dimensional array of square cells which represent a pixel-sized part of the land and contain all the propagation data in the form of variables. The grid can be divided into more cells according to each individual cell size, but for a more clear representation of the landscape a one pixel sized cell was implemented, which also enables a one-by-one correspondence between the fire-grid and the height data array.

So, once the fire-grid is created, there already are information about the height and the exact x,y coordinated available for each cell. However, the only useful piece of information that comes out of the height data is about the slope of the land, which is not enough on its own to properly simulate the fire spread. More information regarding the vegetation type and density of each cell as well as the wind speed and direction are needed. Also, since a cellular automaton is used, the initial state of each cell and the transition rules have to be defined. In this approach, though, there is no apparent way to extract these data from the surface of the tangible GIS, because there are no indications on the sand to imply, for example, the direction of the wind or the type of the vegetation for parts of it.

All these data have to be estimated through the height data that are stored for each cell. So, at first, the information regarding the type and density of the vegetation of each cell had to be defined. To do that three distinct vegetation types and density types were determined. The vegetation types were sand/rocks, grass/agricultural and tress and the density types were sparse, normal and dense. Each of these types were assigned to each cell of the fire-grid according to their height by using Algorithm 3, which utilizes the thresholds that were defined at Section 4.3.3.3 to differentiate each layer of texture/landscape type. The wind direction and speed don't have to be defined for each cell separately, since they are applied to the grid as a whole. However, the method that was used to define these two elements will be discussed later on.

It is pretty clear by looking at the algorithm that the only parts of the tangible GIS that can be considered flammable, are those whose height corresponds to the grass land type, since that is the only layer that a fire can spread to. All the other layers (water,sand,rocks) are not flammable, since it is assumed that there is no type of vegetation on them. This will not give a very accurate image of the vegetation of the landscape created through the height data, because of the limited amount of types of vegetation that can be simulated through assumptions based on the height of each cell. However,

Algorithm 3 Vegetation and States Initialization Algorithm

```
 1: if $height \geq 0$ and $height \leq GrassThreshold$ then
 2:     $vegType \leftarrow sand/rocks$
 3:     $vegDensity \leftarrow sparse$
 4:     $cellState \leftarrow 1$
 5: else if $height \geq GrassThreshold$ and $height \leq RocksThreshold$ then
 6:     $vegType \leftarrow grass$
 7:     $vegDensity \leftarrow normal$
 8:     $cellState \leftarrow 2$
 9: else if $height \geq RocksThreshold$ then
10:     $vegType \leftarrow sand/rocks$
11:     $vegDensity \leftarrow sparse$
12:     $cellState \leftarrow 1$
13: end if
```

as experiments showed, the fire spread simulation is accurate enough to give a fairly good representation of the fire's behavior, that can be used later on finding a minimum cost path to it.

After each cell is allocated with a vegetation type and its density, the only thing that is left is to specify the states in which the cells can transition to and the rules according to which these transitions will happen. These states are basically the states that a real life flammable or inflammable object can be in. So, the possible states that a cell can be in are:

- State 1: The cell is inflammable,which means that the fire cannot spread to it.
- State 2: The cell is flammable but it is not burning yet.
- State 3: The cell is burning.
- State 4: The cell was flammable but it is burned out.

The state of each cell is initialized as illustrated on Algorithm 3 and can be changed at any given time according to the propagation rules. The purpose of these rules is to dictate not only whether a cell itself should transition to another state but also if the fire should propagate to the neighboring cells. Once all the cells are initialized, the fire-grid can be depicted as a four layered matrix, with each layer containing information about different factors (state, vegetation type, vegetation density, slope) that affect the fire spread. The propagation rules take into account only the layer that corresponds to the state of each cell and they are applied every discrete time step T to the I,J elements/cells of that matrix's layer as shown below:

- Rule 1: If the cell is inflammable then it should remain in its current state since the fire cannot propagate to it.

```
1: if $state(i,j,t) = 1$ then
2:     $state(i,j,t+1) = 1$
3: end if
```

- Rule 2: If the cell is burning then at the 8th next time step it will be burned out.

```
1: if $state(i,j,t) = 3$ then
2:     $state(i,j,t+8) = 4$
3: end if
```

- Rule 3: If the cell is burned out then it will stay the same.

```
1: if $state(i,j,t) = 4$ then
2:     $state(i,j,t+1) = 4$
3: end if
```

- Rule 4: If a cell is burning at the current time step then the fire can propagate to any of its neighboring cells (i ± 1, j ± 1) with probability p_{burn}.

```
1: if $state(i,j,t) = 3$ then
2:     $state(i \pm 1, j \pm 1, t+1) = 3$
3: end if
```

4.4.2.2 Calculation of Propagation Probability

As already mentioned multiple times, the way a fire will spread depends on various factors, the most important of them being the vegetation of the land (both type and density), the wind and the slope of the landscape. Each of these factors contribute to the fire's behavior in a different way and with different efficacy. Since, the implemented model of the fire spread is basically a mathematical one, all these factors have to be translated into numbers or equations that will fit into our mathematical model. A smart way to do this is by using probabilities. Specifically, the probability that needs to be calculated is the $\mathrm{p_{burn}}$ probability, which dictates whether the fire will propagate to a neighboring cell or not. This probability can be calculated as a combination of probabilities that derive from each of the factors, as show in Equation 4.1:

$$\mathbf{p}_{burn} = \mathbf{p}_h(1 + \mathbf{p}_{veg})(1 + \mathbf{p}_{den})\mathbf{p}_w\mathbf{p}_s\,, \tag{4.1}$$

where p_h denotes the constant probability that the fire will spread to one of the neighboring cells no matter what the other factors' values are. p_{veg}, p_{den}, p_w and p_s are the fire propagation probabilities that depend on the type of vegetation, the density of vegetation, the wind parameters and the slope respectively. The purpose of the p_h probability is to be multiplied to all these probabilities, in order to obtain the corrected probability that takes into account all the aforementioned factors.

However, each factor have a different way of affecting the propagation probability, which in turn means that each of these distinct probabilities have to be calculated separately.

4.4.2.3 Vegetation Type and Density

So, the first two factors are the land's vegetation type and vegetation density, which are represented by the probabilities p_{veg} and p_{den} respectively. In this approach, three distinct vegetation types were defined for sand/rocks, grass and trees, while the vegetation's density was also clustered into three categories, from sparse to normal and finally to dense vegetation. In the case of vegetation though, all that needed to be defined was the probability of each of the distinct types, since these two factors have no internal factors that can affect them. The values of the probabilities that were assigned to each of the probabilities were chosen empirically and are shown in Table 4.1 and Table 4.2.

Category	**Type**	**Probability**
1	Sand	-0.3
2	Grass	0
3	Trees	0.4

Table 4.1: Probability values for the vegetation types

Category	**Density**	**Probability**
1	Sparse	-0.4
2	Normal	0
3	Dense	0.3

Table 4.2: Probability values for the vegetation density

4.4.2.4 Wind Direction and Speed

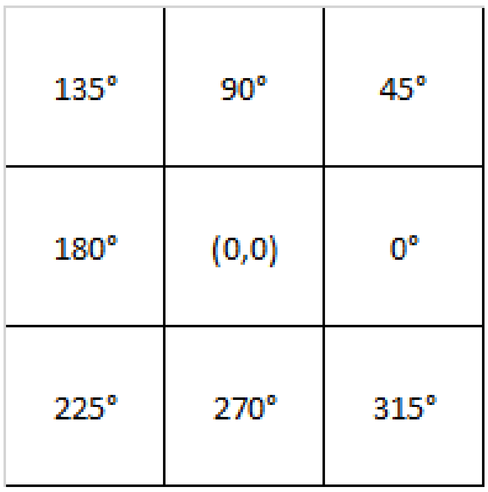
Once, the probabilities corresponding to the vegetation factors are defined, the next element that affects the fire spread is the wind. The wind can affect the behavior of the fire both through its direction and through its speed and is one of the most important factors in dictating the direction towards which the fire will propagate. Sometimes wind can also aid in taking the fire out, since strong winds can force the fire to propagate in areas with low fuels or high moisture, which can lead to the fire burning out or being extinguished easier.

The equation that was implemented in A.Alexandridis' approach was an empirical one that was derived from various other empirical relationships that have been suggested throughout the years to model the effects the wind can have on the rate of a fire spread. The reason that this equation was chosen is due to its flexibility and its perfect fitting to our implementation. Specifically, the probability that represents the effects of the wind speed and direction on the fire propagation is calculated through the following two equations:

$$p_w = \exp(c_1 V) f_t \,, \tag{4.2}$$

$$f_t = \exp(V c_2 (\cos(\theta) - 1) \,. \tag{4.3}$$

c_1 and c_2 are constants that can be determined based on relevant to them parameters that have be reported in other approaches [59, 55], while θ is the angle between the direction of the fire propagation and the direction of the wind. Although, θ can basically take any continuous value from 0 to 360°, in our approach a simpler version of that was implemented. Specifically, θ was calculated according to the basic directions of both the fire propagation and the wind, not according to the exact angle between the two directions. This means that if , for example, the direction of the wind is North then the θ for the cell that is South of the cell from which the fire will spread will be 180°. A full example representation of θ allocations can be seen on Figure 4.14.



135°	90°	45°
180°	(0,0)	0°
225°	270°	315°

Figure 4.14: θ values of neighbors (wind direction:East)

However, in our case there is one more problem that needs to be tackled. In A.Alexandridis' approach the information about the wind's initial direction and speed and how those two changed throughout the wildfire event in Spetses, were all known beforehand. This is not possible in this approach because the implementation is based on the live feed of the Kinect sensor, so a way to define these two factors at any given moment through the frames captured from the sensor's cameras has to be found. But, the methods that were used to extract the information for these two factors will be discussed later on.

4.4.2.5 Ground Elevation/Slope

Last, but not least, the final factor affecting the way the fire will propagate is the slope of the land. The common approach when it comes to slope is that fire spreads easier and faster when it travels upwards, meaning that when a cell's slope regarding to its adjacent cells is positive then the fire is more likely to spread to that cell. This assumption though has to be mathematically modeled in order to be applied to the fire spread model. This can be done by calculating the probability (p_s) that models the effect of the slope to the fire propagation.

An equation for modeling the effect of the ground's elevation (slope) on the fire's behavior have already been introduced in [55]. This equation is the following:

$$R_s = R_{0s} \exp(\alpha \theta_s) \,, \tag{4.4}$$

where R_{0s} is the spread rate when the slope is equal to zero, θ_s is the slope angle of the patch and α is a constant that can be adjusted from experimental data. Through Equation 4.4 the probability that models the effect of the slope on fire propagation can be derived:

$$p_s = \exp(\alpha \theta_s) \,. \tag{4.5}$$

As mentioned α is a constant so it doesn't have to be calculated through an equation, but the slope angle θ_s needs to be calculated and more specifically it needs to be calculated in two different ways depending on whether the two neighboring cells of the square grid are adjacent or diagonal to the cell from which the fire will spread from. So, the slope angle for adjacent cells reads:

$$\theta_s = \arctan(\frac{E_1 - E_2}{l}) \,, \tag{4.6}$$

and for diagonal cells:

$$\theta_s = \arctan(\frac{E_1 - E_2}{l\sqrt{2}}) \,, \tag{4.7}$$

where E_l and E_2 are the values of the elevation of the two cells (the adjacent/diagonal and the burning cell) and l is the length of the square side.

4.4.2.6 Constant Values Calculation

Most of the variables that appear on the various probability equations that were introduced above, can either be defined empirically or they can be extracted through the frames that are acquired by the Kinect sensor. Specifically, the values of the probabilities that model the effects of the vegetation were chosen empirically and are shown on

Table 4.1 (vegetation types effects) and Table 4.2 (density types effects), while parameters like the wind direction and the elevation are based on the live feed that is taken from the sensor.

What is left are the parameters used on these equations as constants, meaning the constant probability p_h, the slope coefficient α and the wind coefficients c_1 and c_2. In [16] these constants were determined by wrapping around the simulator a non-linear optimization technique, setting as objective the minimization of the difference between the number of the predicted burned cells and those that were actually burned during the wildfire in Spetses.

In our case, this cannot be applied since the simulation is not based on a wildfire event that has already occurred. On the contrary, our implementation is based on a real-time image of the tangible GIS and so does the fire simulation. So, despite the fact, that the fire simulator that was created for the Spetses event fits perfectly to our implementation, there are some parameters that need to be defined differently. The way this situation was approached was through experiments. At first, the optimized values that were used on [16] and are shown on Table 4.4, were also applied to our implementation. The results of that were quite appealing, as the simulation of the fire spread was working as intended.

Parameter	**Value**
p_h	0.58
α	0.078
c_1	0.045
c_2	0.131

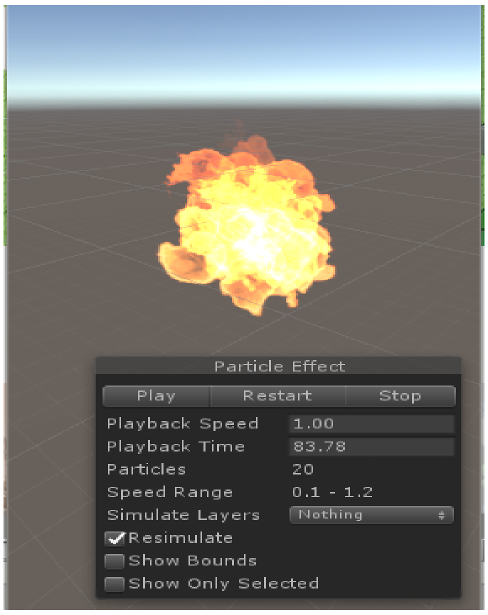
Table 4.3: Constant parameters from Alexandridis Approach

However, various adjustments were made to these values in order to test whether the results would be different or not and in the end it proved that the changes happening to the fire spread simulation were close to none. The last experiment that was carried out, was removing the constant parameters completely, since their use was to adjust the final results of the simulation so that they were closer to the real outcome of the wildfire. That, though, resulted in a less realistically looking fire spread model, so the final values that were assigned to the constant parameters are displayed below.

Parameter	Value
p_h	0.52
α	0.15
c_1	0.045
c_2	0.0491

Table 4.4: Final values for the constant parameters

4.4.3 Fire Simulation Visualization



Playback Speed 1.00
Playback Time 83.78
Particles 20
Speed Range 0.1 - 1.2
Simulate Layers Nothing
✓ Resimulate
Show Bounds
Show Only Selected

Figure 4.15: The fire model

After all the experiments and definitions are made, all that is left is visualizing and testing the simulator as a whole. To do that, the starting point of the fire had to be declared first. The initial approach to this, was to use a "invisible" gameobject placed on the mesh created to visualize the surface of the tangible GIS. This gameobject was a cube whose Mesh Renderer was disabled so that it doesn't interfere with the rest of the scene both physically and visually. Through this gameobject's world position on the scene that was created in Unity3D, the coordinates of the cell on which the fire starting point will be, can be extracted. This is basically a translation of the x and z coordinates of the gameobject's position on the scene to the x and y coordinates of the fire grid and can be done by calculating to which percentage of the width and height of the grid the x and z coordinates values are, as done in [1]. This means that for the x coordinate, if the object is on the far left of the grid it will have a percentage of 0 and if it is in the middle it will have a percentage of 0.5. The algorithm that was used to implement that is shown below:

Once the starting point is approximated the state of the cell which corresponds to that point, is changed to the burning state (state No3) and the fire propagation simulation begins. But, in order for the fire to be visualized in our scene, a fire gameobject

Algorithm 4 Algorithm for extracting grid position out of world position

```
1: $percentX \leftarrow (gameobject.x + gridWidth/2)/gridWidth$
2: $percentX \leftarrow (gameobject.z + gridHeight/2)/gridHeight$
3: $cellX \leftarrow int(gridWidth * percentX)$
4: $cellY \leftarrow int(gridHeight * percentY)$
```

has to be created first. In this approach, in order to visualize the fire, the flame particles that could be acquired from Unity3D Asset store were used. More specifically, the flame gameobject was created by using some parts of the flame particles that were available and then it was turned to a prefab. The final fire model that was used as the prefab is shown in Figure 4.15.

Prefabs in Unity3D are basically a saved form of a gameobject that can be loaded to the scene as many times as the user wants. Saving the created flame gameobject as a prefab, can enable multiple uses of it to create a fire. So, visually the fire propagation works like that: first, a flame gameobject is instantiated (through its prefab) at the position in the scene that corresponds to the starting cell of the fire grid.

Then, the propagation probability p_{burn} is calculated for every neighboring cell of the starting one and if this probability exceeds a threshold defined by the user (in this thesis this threshold is equal to 0.5), then the fire will spread to the cell for which that probability was calculated. This spreading is visualized by basically instantiating another flame gameobject at the position of the cell that the fire will spread to. This procedure goes on until there are no more cells that can be burned or the fire simulation is halted manually.

Later on in this thesis, a new method for acquiring the starting point of the fire will be discussed, so the cube object that was created in this section will no longer be needed. No matter the case, a snapshot of the simulation so far can be seen in Figure 4.16. For this specific scenario, the wind direction was set to east and its speed to 10 m/s. The rest of the parameters needed for the simulation were extracted through the procedures that were discussed in this section.

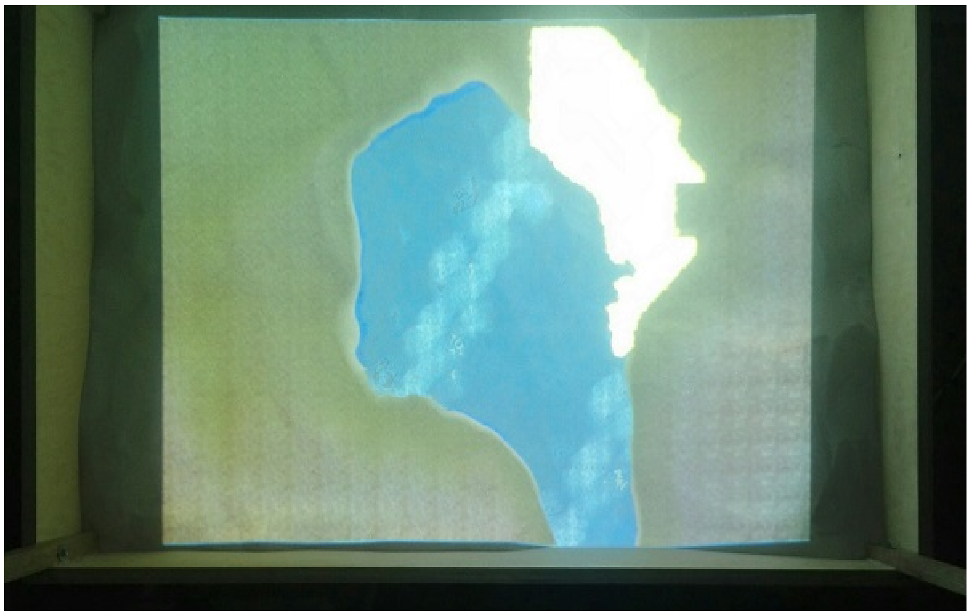


Figure 4.16: A snapshot of the fire projected on the sand

4.5 Minimum Cost Path-Finding

This far on this thesis, the two basic parts of a fire simulation - the landscape and the fire spread model - have been implemented. However, the purpose of this project is not just to create another fire simulation which can be reconfigured in real time, but to implement a tool/application that any firefighting brigade could find both helpful and educative. So, for this to happen new components, that can prove to be useful on real life firefighting scenarios, had to be added to the simulator. Some of these components can be the evaluation of the minimum cost path that leads to the fire front, the representation of evacuation plans, if, for example, a city or village is located near the fire or the allocation of the agents of the firefighting force in such a way so that the fire is extinguished in the best possible way.

In this approach, a bigger emphasis was given to the component of the path-finding. Specifically, our goal was to calculate the path that would take all the fire-fighters/agents from their starting point to the fire front in the least amount of time and effort. There are many algorithms that can be used in order to implement that, however, in our case two algorithms were tested: the weighted A* and the Value iteration algorithms. But, before these algorithms can be applied in the simulator, their unique parameters had to be defined first.

This, though, is not the only problem that had to be tackled. Both of the aforementioned algorithms cannot be directly applied onto the 3D model that is created so far, but both of them can be applied on grids. This fits perfectly to our implementation, because a similar technique has been used at the previous Section 4.4, where a fire grid was created in order to model the fire spread through a Cellular Automaton. The same grid could be used in this case as well, by storing the information needed for the algorithms to work in its fire cells. However, for clarification purposes that was not the approach that was followed. Instead, in order to isolate the information needed for the fire simulation from those needed for the path-finding, a new grid was created.

4.5.1 Grid Implementation

This time, though, the implemented grid was mostly based on the work of Sebastian Lague [1]. In his approach, Lague intended to create a path-finding application via Unity3D, by creating a grid that could be applied to any flat plane gameobject or mesh and on which the A* algorithm could be used for finding a path between two points of it. This grid is very similar to the fire grid that was created for the fire model and is consisted of nodes which carry all the data that are essential for the algorithm to work. As stated in Section 2.5.1 the A* algorithm requires a cost (G) and a heuristic (H) function in order to evaluate the minimum cost path. However, in this case a few more variables were added which corresponded to the Nodes' coordinates, their respective position in the world scene and an indicator on whether they are walkable or not.

The procedure to create the grid is very similar to the one followed on the previous section. Each of the nodes has a radius which corresponds to its size and determines the number of nodes that the grid will be divided into. In our case, the Node's radius was set to 0.5, so that the size of each Node was equal to the size of a pixel. That way, each cell can be associated with its corresponding pixel of the depth image that is obtained from the Kinect sensor. This also means that the coordinates of each cell correspond to the x and y coordinates of each value in the depth (or the height) data array, so that these two are directly connected to each other. The grid is generated by creating a Node for every pixel of the depth image, so in the end the dimensions of the grid match the dimensions of the cropped image that is acquired after the calibration procedure (in Section 4.2). Also, since the same process was implemented for the fire grid, there is a one-by-one correspondence between the height data array, the fire grid and the path-finding grid.

As far as the walk-ability of a Node is concerned, there are two cases. The first case, which is the one Lague implemented in his work, is when obstacles appear in the form of gameobjects (e.g cubes). This can be used in our implementation too, to represent obstacles that can block the path of the firefighters/agents, like, for example, boulders or fallen trees. The second case, which is the one that our approach mainly focused on, is when a Node's walk-ability is determined through its corresponding height value or layer type. Specifically, a height threshold was defined, so that all the Nodes whose height was higher than this threshold would be determined to be unwalkable, while the Nodes whose height was corresponding to the height of the water layer would also be considered unwalkable. Basically, the Nodes that are unwalkable are the ones corresponding to sea or lake parts of the map, as well as, the ones correlating to rocky mountains.

4.5.2 Weighted A* Algorithm Implementation

Once the aforementioned procedure is completed, the grid is built in the form of a two dimensional array of Nodes, each of which contains all the information needed for the A* algorithm. However, four new variables had to be added, that resembled to the G-function cost, the Heuristic function cost, the F cost, which basically is the addition of the two aforementioned costs, and a pointer that indicated the parent of each Node. Also, in our approach, each of the Nodes had to have some kind of indicator that showed whether this Node was a target Node or the Node where an agent is located. These indicators were two additional variables, one for the case of the target Node and one for the agent Node. Specifically, for our case, all the Nodes that corresponded to the current burning cells of the fire grid (through the one-by-one correlation) were also considered to be target Nodes.

Lastly, since the A* algorithm is a weighted one, the "weight" of each of the Nodes had to be defined as well. This was done through the form of a movement penalty for passing through that Node. These movement penalties were determined according to the height of the corresponding Node much like how the vegetation types and densities were assigned to the fire cells. In Algorithm 5 it is shown how the movement penalties and the walk-ability were allocated according to the layer type of each Node, which in turn corresponds to its height value.

Algorithm 5 Movement Penalty and Walk-Ability Allocation

```
1:  if $type = Water$ or $type \geq StonyGround$ then
2:      $movementPenalty \leftarrow maxPenalty$
3:      $waklable \leftarrow false$
4:  else if $type = Sand$ then
5:      $movementPenalty \leftarrow maxPenalty/2$
6:      $waklable \leftarrow true$
7:  else if $type = Grass$ then
8:      $movementPenalty \leftarrow 0$
9:      $waklable \leftarrow true$
10: end if
```

The purpose of the implemented A* algorithm is to find the minimum cost path through the f cost of each Node and connect the Nodes forming the path from an agent Node to a target Node through their parent pointers. This was carried out by creating two data structures, a heap for all the new Nodes that are being studied at the moment (openSet) and a hash set that contains all the Nodes that have already been studied (closedSet). The heap structure was chosen specifically, because of its unique function to immediately sort the nodes that are inserted to it according to their f cost. That way the first item of the heap will always be the Node with the lowest f cost, thus by studying these Nodes first the minimum cost path will eventually be estimated. The procedure according to which this is implemented is shown in Algorithm 6.

It is clear, by studying the algorithm, that the heuristic function that was used in this case was the distance between the neighboring Node and the target Node, assuming that the position of the target Node is known. This way, as the position of the Nodes gets closer to the position of the target Node, the output values of the heuristic function will get lower, meaning that the cost of transitioning to the corresponding neighbor Node will also get lower, until the target is reached. As for the movement

Algorithm 6 Implemented Weighted A* Algorithm

```
 1: add start Node to openSet
 2: while openSet not empty do
 3:   currentNode ← remove first Node from openSet
 4:   add current Node to closedSet
 5:   if currentNode = fireNode then
 6:     done
 7:   end if
 8:   for each walkable neighbor Node of currentNode not in closedSet do
 9:     Cost ← movement cost from current to neightbor Node
10:     if cost < neighbor'sGcost then
11:       neighbor's G cost ← Cost
12:       neighbor's H cost ← distance from neighbor to target
13:       neighbor's parent ← currentNode
14:       add neighbot Node to openSet
15:     end if
16:   end for
17: end while
```

cost of moving from the current Node to one of its neighboring ones it reads:

$$Cost = currentNodeGcost + distance + penalty\,, \tag{4.8}$$

where the distance is calculated as such:

$$heightDifference = currentNodeHeight - neighborNodeHeight\,, \tag{4.9}$$

$$distance = \sqrt{(currX - neighX)^2 + (currY - neighY)^2 + heightDifference^2}\,, \tag{4.10}$$

where *currX*, *currY* and *neighX*, *neighY* refer to the x and y coordinates of the current and its neighboring Node respectively.

Once the A* algorithm converges, the path that leads from the starting Node to the target can be retraced through the Nodes' parent Nodes. Specifically, starting from the target Node's parent and moving on until the starting Node is reached will result in deciphering the minimum cost path as a sequence of Nodes. This happens due to the fact that the Nodes that are added in the Heap structure (openSet) are sorted according to their f cost. This way, the Node with the lowest f cost will always be studied first, so in the end the Nodes that form the path will be the ones with the lowest possible costs.

Overall, the weighted A* algorithm is very efficient in the perspectives of both the time it requires to find the minimum cost path and its capability to find an optimized path. Specifically, the time that is normally required for finding the optimal path ranges between 10-30 milliseconds, while the path that is calculated is usually the expected one. However, there are some scenarios that can prove challenging for the A* algorithm. One of these scenarios is where the path can lead to certain arc shaped obstacles (traps), while the optimal path would be to avoid them. This happens because the heuristic function is based only on the virtual distance between each Node and the target Node, so unless the G function suggests otherwise the path will lead to Nodes that are blocked by obstacles but are closer to the target.

This issue can be tackled by implementing a smarter heuristic function, but is our approach the Value Iteration algorithm was also implemented and a comparison between the two was made.

4.5.3 Value Iteration Implementation

Regarding the Value Iteration (VI) algorithm, a few modifications had to be made to the information that needed to be stored in each Node. Specifically, all the data that corresponded to the coordinates, the height and the type (Fire, Agent) of each Node were conserved, but the VI algorithm does not require any information as to the output values of the functions used in A* (G and H). On the other hand, VI requires, as stated in Section 2.5.2, an initial utility value for every Node of the grid that it is going to be applied on and a strategy according to which the minimum cost path will be evaluated. So, in the end, two new variables were defined for each Node, one for its utility value and one pointer indicating the neighboring Node that has the lowest utility and thus is the best option for the agent to transition to.

So, once all the important data needed for the algorithm were defined, the grid had to be initialized. This initialization procedure is similar to the one followed for the A* algorithm, apart from the fact that the starting utility of every Node had to be determined as well. This time, though, the initial utility of the Nodes is not based on their corresponding height value. Instead, every Node is given the maximum possible utility (defined by the user) unless it is a target Node, so its initial utility is set to zero. These values could be opposite, meaning that the target Nodes' initial utility would be set to maximum and for all the other Nodes to zero, but the final result would have no difference. As for the walk-ability of each Node it was allocated in the same way, as shown in Algorithm 5.

The way the VI algorithm works has a very distinct difference compared to the A* algorithm, that lays on how the grid is scanned. Particularly, the A* algorithm starts its searching process from the starting Node and then proceeds to its best neighbor. This means that the walkable part of the grid doesn't have to be scanned entirely, since a path can be evaluated in the first try (with no backtracking). However, the VI algorithm scans the entirety of the grid, calculating the utility for every walkable Node that is not a target and evaluating which is the best action for the agent to take when it is on a specific Node. Also, another huge difference between the two algorithms, is the fact that A* can only find the path towards a single target, while the VI evaluates all the paths that lead to every single target Node that might exist in the grid at the given moment.

But, before the VI algorithm that was used in this thesis was defined, our case's problem had to be translated to a Markov Decision Process (MDP). A MDP is defined by a finite set of states (S) and actions (A), a probabilistic function that outputs the probability of an action succeeding ($P_a(s, s')$) and a reward function ($R_a(s, s')$), which determines the reward of transitioning from a state to another because of a specific action (a). In this approach, the set of states consists of all the walkable Nodes of the grid, while all the possible movements of the agent/firefighter comprise the set of actions. However, is our approach, the stochastic part of the MDP was completely left out, since the agents are assumed to be firefighters, which in turn means that there is no possibility for their actions to fail. As for the reward function, it was replaced with the movement penalties of each Node, meaning that the reward for moving from a Node to another is the movement penalty for moving to that particular Node.

So, as in the A* implementation, the purpose of the VI algorithm is to find the minimum cost path. Since, during the initial allocation of the utilities the target Node is given a zero utility, the goal is to calculate a path that leads to it by "spreading" the target Node's low utility throughput the grid. This "spread" procedure occurs in every iteration of the algorithm, starting from the target Node and then in each iteration moving on to its neighbors and their respective neighbors and so on, until every walkable Node has been studied. This means that once the VI algorithm converges all the walkable Nodes' utilities will have been calculated and the path can be evaluated through each Node's pointer to their neighbor with the lowest utility.

However, in order for the VI to converge a few more parameters have to be defined. These are the tolerance of the algorithm, which determines when the algorithm is assumed to have converged, the γ constant, which in our case was set to one, and the maximum number of iterations that can occur. The max iterations are defined for the unlikely case that the algorithm cannot converge, due to the incapability of the algorithm to evaluate a path, because, for example, a mountain range (which corresponds to a series of unwalkable Nodes) blocks the path to the target. The algorithm that was used for this process and all of its parameters are presented below:

The utilityMax value is this algorithm corresponds to the maximum difference between the current utility of a Node and its new calculated utility. This value gets lower with every iteration, until it is lower than the predetermined tolerance of the algorithm. Basically, this will happen when there are no more changes on the utilities

Algorithm 7 Implemented Value Iteration Algorithm

```
1: while not $converged$ and not $maxIterations$ do
2:   $utilityMax \leftarrow 0$
3:   for $each\ walkable\ non\ target\ Node$ do
4:     $previousUtility \leftarrow Node's\ Current\ Utility$
5:     $Node's\ Utility \leftarrow MAX\ Utility\ Value$
6:     for $each\ walkable\ neighbor\ Node\ of\ current\ Node$ do
7:       $new\ Utility \leftarrow CalculateUtility$
8:       if $new\ Utility < Node's\ current\ Utility$ then
9:         $Node's\ Utility \leftarrow new\ Utility$
10:        $Node's\ nextBest \leftarrow Neighbor\ Node$
11:      end if
12:    end for
13:    $difference \leftarrow |\ Node's\ Current\ Utility - previousUtility\ |$
14:    if $difference > utilityMax$ then
15:      $utilityMax \leftarrow difference$
16:    end if
17:  end for
18:  if $utilityMax < tolerance$ then
19:    $converged \leftarrow true$
20:  end if
21: end while
```

of the Nodes, which means that the minimum cost pathways to the target have been evaluated. As for CalculateUtility, it is the function that returns the prospective utility value for the current Node. The equation through which this value is calculated reads:

$$utility = neighborUtility + distance + movementPenalty\,, \tag{4.11}$$

where distance is calculated through Equation 4.10, exactly the same way as in the A* algorithm.

The VI algorithm compared to the weighted A* is far more robust and consistent in evaluating minimum cost paths. Also, it has an advantage on A* regarding the fact that once the VI algorithm converges the path to the target doesn't have to be recalculated, while on A* the procedure has to be repeated. On the contrary, once the VI algorithm converges all the possible paths to the target have already been evaluated no matter the starting point. However, it is obvious that such a procedure requires more time and computational power, which doesn't make it essential for a real-time adjustable terrain, since it might delay the whole simulation process until all the calculations are completed. But, through experimentation it proved that the increased execution time of the algorithm was not causing major issues and the little delays that occurred could be overseen due to the reliability of the algorithm.

4.5.4 Updatable Terrain and Grid

So far, only the use of these two algorithms on stable, non-changing grids has been studied. In this approach, though, that is not the case. In fact, the whole purpose of this thesis is to simulate the firefighting - and thus the path-finding - scenarios in real time, which in turn means that the grid on which the algorithms are applied will constantly change. These changes appear in the form of altering height data or walkability and changing target or starting points. When changes like these happen the algorithms have to reevaluate the minimum cost paths, due to the fact that their cost is tightly associated to the height data that are obtained from the Kinect Sensor.

In the case of the weighted A* algorithm the reevaluation procedure is an simple one, because of the way the path is defined. Specifically, all the values that are used in the algorithm, like the G-function cost and the heuristic function output are always recalculated every time the path has to be evaluated. The only parameters that change and affect the algorithm are the starting points, since the first Node that is studied might differ from the previous ones. The height data only affect the outcome of the algorithm, as the previous path might have been blocked off or a new path might be the minimum cost one. Otherwise, the algorithm always checks the neighbors of the current Node and proceeds to the one with the lowest cost (F-cost) until the target is reached, and since this procedure is a relatively fast one it can be repeated every time the grid changes.

On the other hand, the VI algorithm cannot adopt to the updated grid that easily. This happens because the utilities that are stored in each of the Nodes have been calculated based on the previous height data. This means that when the grid changes and the algorithm does not run again, the evaluated path may lead to false target Nodes or may pass through unwalkable terrain. This means that all the utility values of the grid have to be reset and the algorithm has to run again every time the generated landscape/mesh updates. However, this can prove to be a very time consuming procedure and even more computationally challenging.

One way to avoid this, is by semi-updating the utility values of the Nodes. The concept behind this is to update the utility values only on the Nodes whose height data changed and leave everything else unchanged. This way, when the VI algorithm runs again it will converge much faster than if the grid was fully reset and the path that will be evaluated afterwards will be the correct minimum cost one. However, this solution is not the optimal, since experiments proved that sometimes the minimum path is not evaluated correctly, which may lead to false paths or even may cause the simulation application to crash. So, another solution was found that was not as fast as the first one, but it was much more robust.

This time, the approach that was followed was a “mini” reset of each Node every time they were scanned by the algorithm. This can also be seen on Algorithm 7, on lines 4 and 5, where the utility of the Node that is being studied at that moment is reset to its initial value (which is the max utility value). This may at first seem to be like a full reset of the grid. However, since the utilities of the Nodes don’t reset all at once, if one of the neighboring Nodes has the optimally minimum utility on it, it will be “spread” to the current Node whose utility will never change again. The only changes to the optimal utilities, will occur only if the grid changes entirely or if the previous next best neighboring Node becomes unwalkable. This way, the VI algorithm can reevaluate the minimum cost path without taking as much time as it would need for its initial run.

4.5.5 Path Visualization

Once the minimum cost path is evaluated, it has to be visualized in order to be visible in the 3D model that has been created this far. This visualization procedure is quite simpler than the previous two, because all that needs to be implemented is a line that connects the starting Node with the target Node. In Unity3D, this line can be visualized through a line renderer component, which is a graphics asset of Unity3D that is used exactly for what its name suggests: drawing lines. A line renderer in Unity3D takes as input an array of points (two or more) in the 3D space and draws the lines that connect these points to each other. This function fits perfectly to our implementation, due to the fact that every Node has stored information about its position in the 3D space. This way, the path can be visualized by passing the world position coordinates of each Node that comprises the calculated path onto the line renderer.

Before that, though, the starting point or the starting position of our agents has to be defined. This problem was approached in the same way as in Section 4.4.3, where an "invisible" cube gameobject was created and placed at the position where the fire should start from. This time, the cube that was created was used as the position where our agent's 3D model would be instantiated. Specifically, a gameobject prefab, much like the flame prefab, was created and was instantiated on the position of the cube gameobject that corresponded to it. Along with every agent, though, a line renderer had to be created as well, which will correspond to the path that starts from that agent. So, after every agent was created and positioned on the 3D scene, the path-finding algorithm would evaluate the minimum cost path between the agent and the spreading fire and then pass the Nodes' positions to the line renderer in order to highlight that path. Later on in this thesis, a new approach will be followed to find the starting points of both the fire and the agents.

The way these points were passed to the line renderer, though, was not the same for both of the two implemented algorithms. In the first case, the A* algorithm evaluated the path and then could return it in the form of an array of 3D vectors. These 3D vectors are basically the positions in 3D space of each of the Nodes that consist the output path, which means that they can be immediately used by the line renderer as its input 3D space points. However, in the case of the VI algorithm, the output is a fully mapped grid, where every Node has a pointer that if all of them are followed one by one, they will lead to the nearest target. This means that the 3D points that have to be passed to the line renderer, must be taken one by one through the sequence of Nodes from the starting to the target one. So, basically, in order to acquire all these positions multiple loops had to be implements, one for each starting Node, that read through the pointers of each best next Node starting for the starting Node and pass those Nodes' positions to the corresponding line renderer, until a target Node was reached.

Once, the line renderer had obtained the necessary points, the parameters that influence the look of the line can be defined as well. Specifically, the line's width and color were also defined, in order to make the line clearer from a top-down camera view. All the other parameters that could be adjusted had to do with how the line would interact with the lighting of the scene and some other visual related factors.

In this approach, the visualization of the minimum cost path was the last part of the simulation that had to be visualized, so in Figure 4.17 a snapshot of the final projection of the 3D model of the tangible GIS along with all its features is shown.

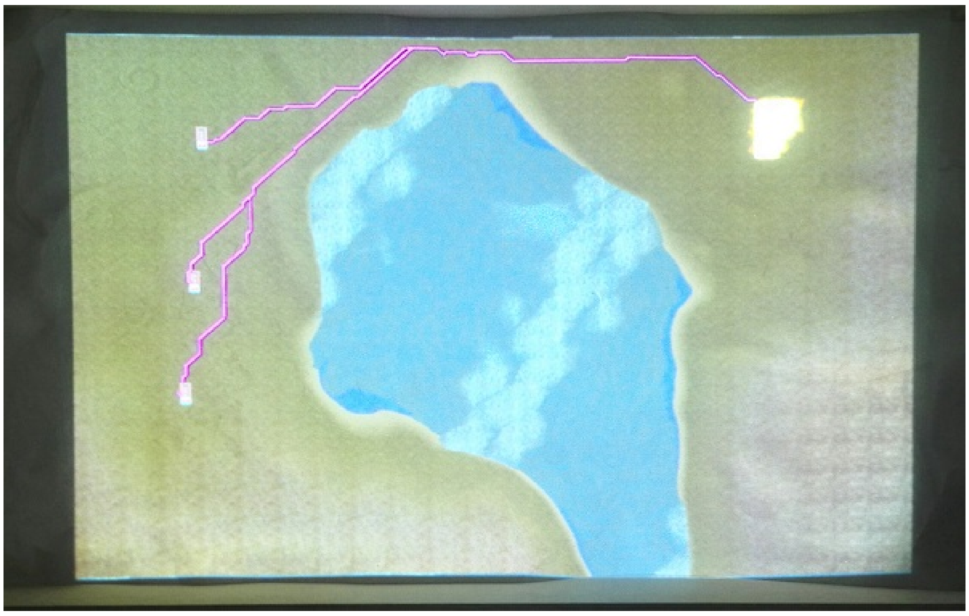


Figure 4.17: A snapshot of the minimum cost path (final look)

4.6 Object Detection

As already mentioned in Section 4.4 and Section 4.5, the starting points of both the fire and the agents are defined through gameobjects that are inserted in the scene and then are destroyed when they are no longer needed. This procedure though did not offer much freedom to the user as to where the starting points would be positioned and was also taking away some of the interactivity of the user with the sand, since the gameobjects had to be created and positioned through the Unity3D editor. Also, the part of the procedure that involved the work done in the editor of Unity3D was relatively a slow one, which in turn was causing issues to the real-time aspect of this project.

So, a new way to define the starting points of the fire and each agent had to be implemented. The solution to this problem, came from the Kinect Sensor and specifically from its color camera. The color image obtained from the Kinect sensor contained not only the top-down view of the sandbox, but also all the items that could potentially be placed on it and since this image could be extracted and edited, object detection techniques could be used on it. This means that items of different colors and shapes can be placed on the sand, in order to be detected through the camera and then be used to define either the starting positions of the fire and the agents or other information that could prove useful to the simulation (e.g. the wind direction and speed).

4.6.1 Setup and Calibration

Before the objects on the sand could be detected though, the object detection tools had to be set up. There are a many applications and techniques available for detecting objects or processing images in order to extract extra information out of them. There are also libraries that contain all the known object detection techniques that can be used on almost every image. One of the most popular of these libraries is OpenCV [21]. These libraries can be imported to any compatible project and all the techniques stored in them are at the disposal of the owner of that particular project.

In our approach, the library that was imported was the EmguCV one [6], which is an OpenCV integration library, so that the OpenCV library can be used on any Unity3D project. This library contains known object detection techniques, like Hough

Circles and Lines transform, which can be used to detect circles and lines respectively in an image. So, in the end, through this library multiple objects of different shape, size and color can be detected on the sand, but specifically for this thesis only three distinct shapes were chosen: circles, rectangles and triangles. Each of these shapes had a different purpose, which will be discussed later.

Once the library and its corresponding plugin were imported on our project, the object detection procedure could be initialized. The ultimate goal of this process is to get the 3D space positions that correspond to the positions of the objects on the color image. So, once the color image is acquired (through the same procedure as in the previous sections) it had to be encoded to a compatible type (JPG or PNG), so that it can be processed. In order to get the 3D space positions, the positions of the objects on the sand and thus the color image had to be extracted first. In this approach, the objects that had to be detected were circular, rectangular or triangular shaped objects and their use was for defining the wind speed and direction, the starting points for the fire and the starting points for the agents respectively.

However, after the first experiments with the object detection on the sand, it was noticed that all the detected objects' positions were a little bit off compared to the real positions of the objects by a fixed offset. This, of course, was creating many issues, since the starting points could be misplaced onto unwalkable sections of the map, like, for example, the fire's starting point could be placed onto a water part of the map. To solve this problem, a second calibration application was implemented, this time, in order to calculate the fixed difference between the detected and the real position of the objects on the sand.

Specifically, the calibration functions in two repeating phases. In the first phase, the user is asked to place a circular object on a random location on the sand. This location is "marked" on the sand by a projected cross on the sand's surface. The user had to place the circular object at the point where the two lines of the cross intersect and then press a key to continue. Once the key in pressed the second phase of the calibration procedure initiates. In this phase, the object gets detected and the detected position in projected in the sand using a different colored cross, as shown in Figure 4.18. The detected position though is the incorrect one, so the offset between these two positions (the point where the circular object had to be placed and the detected one) is calculated for both of their x and y coordinated and is stored for future use. These two phases

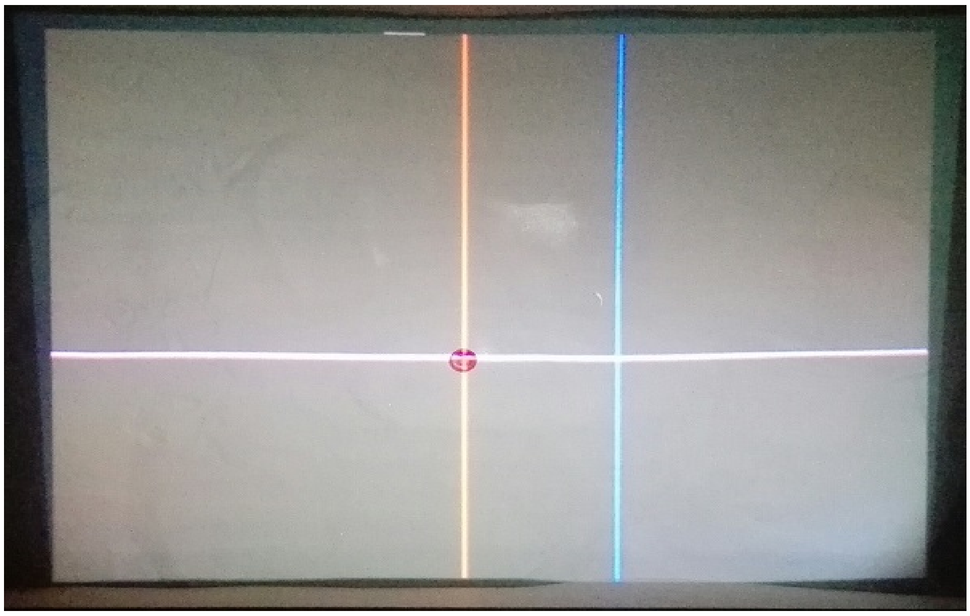


Figure 4.18: A snapshot of the object detection calibration procedure.

are repeated four times for difference random points on the sand and when the whole procedure is over the average offset from all the points is calculated. This way, when the objects are detected on the sand, the stored offset is subtracted from the detected point, so that the final detected position would be the correct one. In Figure 4.19 the difference between the positions that were detected before and after the calibration procedure can be seen (the center of each square that appears in the picture is the detected point).

4.6.2 Starting Points Detection

So, once the detected object's positions are guaranteed to be correct, they can be used to replace the cube gameobjects that define the starting positions of both the fire and the agents in the 3D space. This can be done by translating the detected objects' positions on the color image to points in the 3D space scene and specifically to points on the implemented 3D model of the sand's surface. This translation procedure is basically a mapping between the color image and the depth image of the Kinect Sensor that has already been addressed in the first calibration step (Section 4.2). Basically, the detected points on the color image are mapped to their corresponding points in the

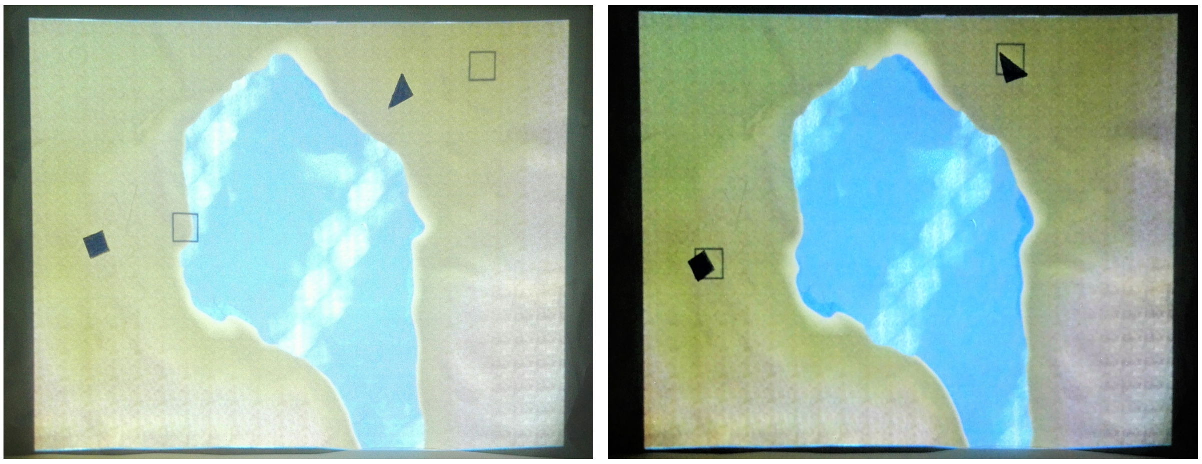


Figure 4.19: Detected positions before (left) and after (right) the calibration.

depth image and due to the way the whole project is implemented, the points of the depth image can be directly translated to 3D space coordinates, since there is a one-by-one correspondence between the depth data and the cells of the grid that was used for the path-finding procedure, whose cells has stored information about its 3D space position. So, connecting the detected positions to their corresponding cell of the grid immediately determines its position in the 3D space.

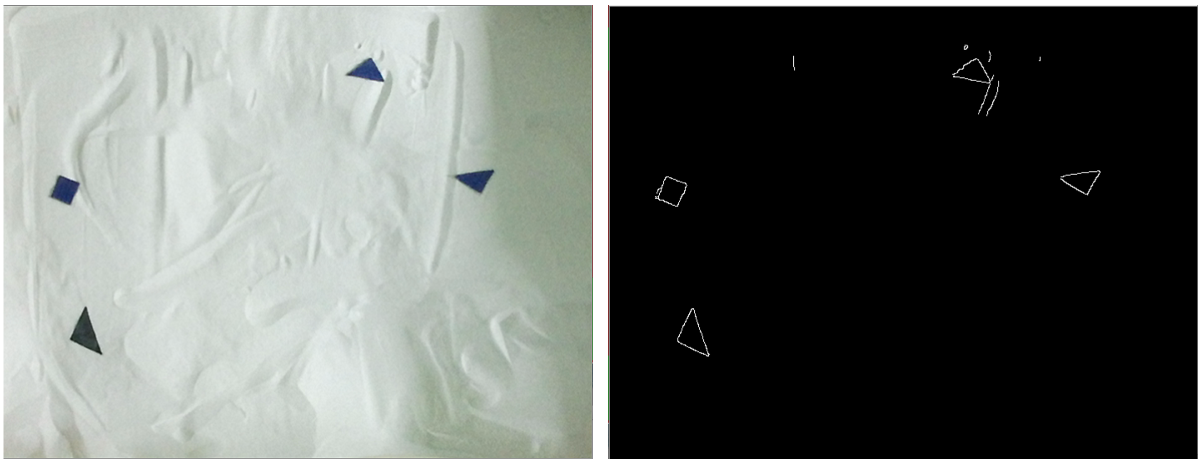


Figure 4.20: Source image (left) and image after Canny filter (right).

The objects that were used to define the starting point of the fire were of a rectangular shape, while the ones used for the agents' starting points were of triangular shape.

The way these two shapes can be detected in a color image is through a combination of a Canny filter and a Hough lines transform. The Canny filter, as mentioned in Section 2.6.1, detects all the edges in an image and filters everything else out, creating this way an image containing lines corresponding to these edges. Through these lines and by using the Hough lines transform, which detects straight lines in an image, triangles and rectangles can be detected as a closed set of three or four, respectively, lines connected to each other. However, in our case, the color image of the sand's surface contains many edges, not only due to the appearance of the objects placed on the sand, but also due to the different shapes that are created from the sand itself. Fortunately, the lines corresponding to the shapes made by the sand are rarely connected to each other in a closed set, which it turn means that they are very rarely detected as triangles or rectangles. An example of the output of the Canny filter on the cropped color image of the Kinect sensor is shown in Figure 4.20.

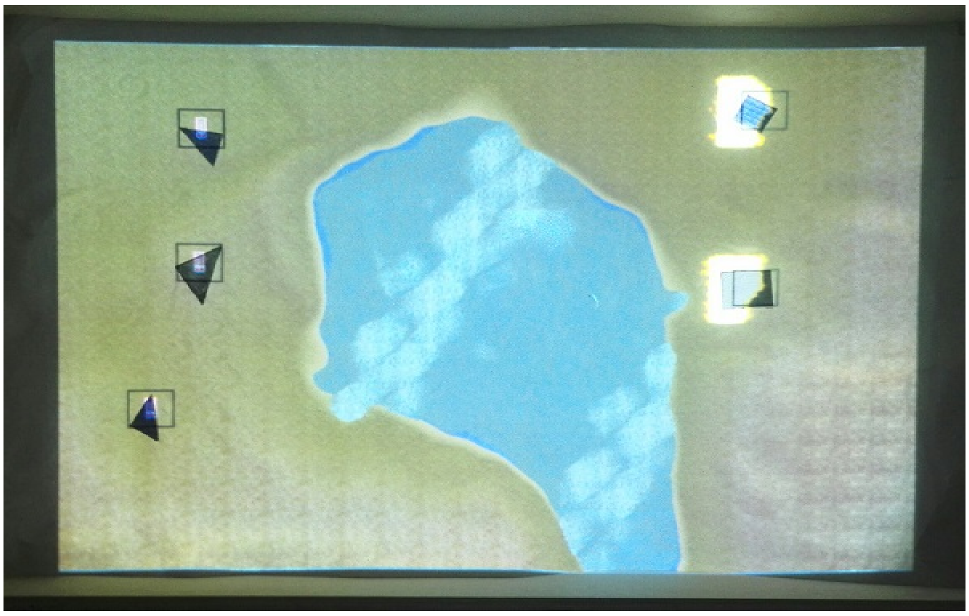


Figure 4.21: Multiple detected triangles and rectangles on the sand and their respective activated features.

In the end, through the attachment of the object detection procedure to our project, the user can determine the starting points for both the fire and the agents, but also the number of fire fronts and agents that the specific simulated scenario will have, since every rectangle detected in the sand corresponds to a starting point for a different fire front, while every triangle defined the starting point of an individual agent/firefighter. This means, that in real time, the user can dictate where there should be fire, where each agent will be positioned and how many of each of those will appear on each scenario, by simply moving the objects around and by adding or removing some or all of them from the sand. An instance of a simulation with multiple detected objects is shown in Figure 4.21.

4.6.3 Wind Speed and Direction Definition

Lastly, there are still two parameters of the simulation that cannot be modified in real time. These parameters are the wind direction and the speed, which, as mentioned in Section 4.4.2.4, are given fixed values that cannot be changed while the simulation is running. This means that there is no way to simulate the changes that may occur during a fire spread and thus the fire simulation lacks that real time altering element. Also, another thing that hasn't been utilized yet is the detection of circular objects, which can be used to tackle the aforementioned problem and provide the simulation with a real-time altering wind direction and speed system to be used to make the fire model look even more pragmatic.

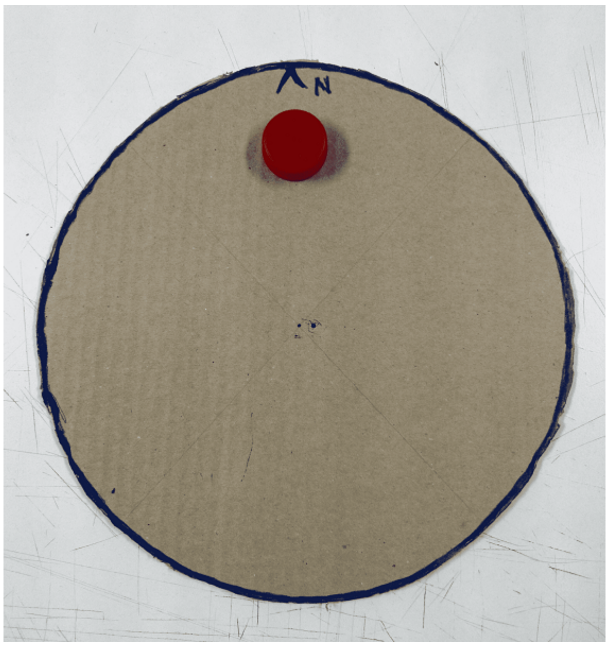


Figure 4.22: The circles system used for the wind parameters.

To implement the real-time manipulation of these two parameters, a system compounded of two circles was created. Specifically, as shown in Figure 4.22, the two circles was of different radius with one of the circles having a significantly bigger radius than the other one. The purpose of this, was to use the bigger of the two circles as a base and the smaller one as a pointer. This way, the pointer circle could be used to dictate both the direction and the speed of the wind parameter, simply by detecting both of the circles and determining the pointer circle's center position in correlation to the base circle's center. However, the two circles system that was crafted was not placed on the sand, because it would block part of sand's surface and with it part of the depth data that were useful.

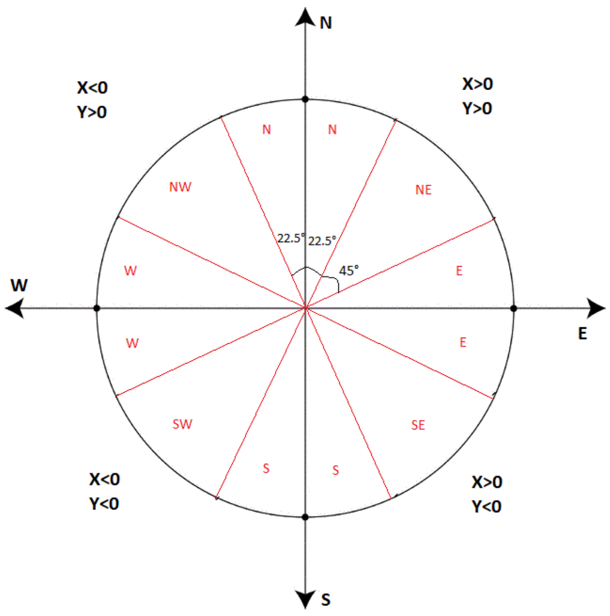


Figure 4.23: Wind directions allocated on the circle based on the angle.

So, the two circles could instead be attached to the exterior part of the sandbox or placed next to it, on a place where the Kinect sensor's camera could see them.This also means that for the circles detection the not-cropped color image had to be used, since the circles system would not be visible in the cropped image. To detect circular objects on the sand only the Hough circles transformation technique had to be used, as shown in Figure 4.24.

Through it all the circled shaped objects could be detected and information about each circle's center point and radius could be extracted. However, the Hough circles transform often detected more circles than the two circles composing the implemented system. This was mainly caused due to the poor definition of the two thresholds that the transform takes as input and the solution that was figured out was applying the Hough transform twice in the same image each time with different thresholds. This way, each of the times the transform is applied, one of the two circles of the system is detected until both of the are.

Once the center points of each circle were detected the position of the pointer circle's (the one with the smaller radius) center in correlation to the base circle's center had to be calculated. This was implemented by subtracting both the x and y coordinates of the pointer's center point from those of the base circle. Then, through these

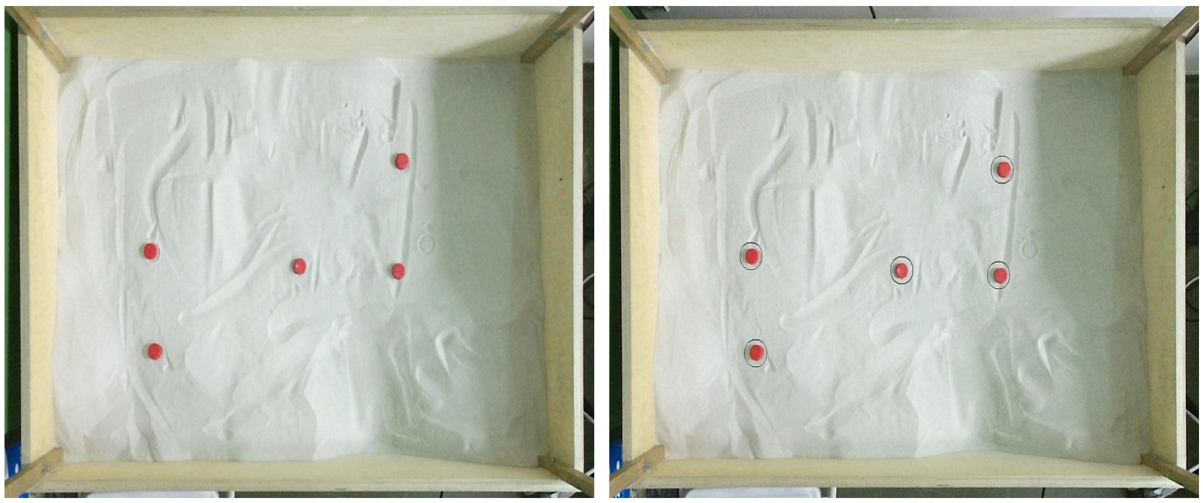


Figure 4.24: Source image (left) and detected circles from Hough Transform (right).

two differences both the angle and the distance between these two points were calculated, which could then be translated to the wind's direction and speed (in m/s). In the case of the wind's direction, the quadrants of the base circle on which the pointer circle was located had to be determined through the two coordinates differences and the precise angle would define the final direction of the wind. In Figure 4.23, a more accurate representation of how the base circle was divided in order for the wind direction to be determined can be seen, where X and Y are the differences between the x and y coordinates of the circles respectively. As for the wind's speed, in this thesis, it was assumed to range from 0 to 28,4 m/s, while the distance of the pointer circle's center from that of the base circle could range for 0 to the radius of the base circle. This means that the distance between the two centers had to be scaled in order to correspond to the wind's speed values range. This was done through the following two equations:

$$centers'\ distance = \sqrt{(baseCircleX - pointerCircleX)^2 + (baseCircleY - pointerCircleY)^2}\,, \tag{4.12}$$

$$wind's\ speed = \frac{center's\ distance}{(\frac{base\ circle's\ radius}{28.4})}\,. \tag{4.13}$$

The wind's direction and speed calculation procedure is repeated before every propagation step of the fire model. This way, when the time comes for the cells to which the fire will spread to be determined, the information about the wind parameters will already be available to be used in the model. This implementation, also, allows the user to modify both of the wind parameters in real time, by simply moving the pointer circular object on a different position onto the base circle.

Chapter 5

Results

In this Chapter we present the results of our work. Throughout the development of the simulation application, various tests were carried out to observe whether the system was functioning as intended or adjustments and refinements had to be made. Specifically, in our case, since the project consisted of several distinct parts, we tested each phase separately, until they were all combined to give the end product.

These distinct test phases of our thesis were:

- **Tangible GIS** : This phase consists of the digitization of the physical model of the GIS (the sandbox) and the generation of its 3D model. Once, the 3D model is created, it is projected on the sand, so that the users can observe and interact with it.

- **Wildfire Model**: In this phase, the wildfire model was implemented and was embedded to the aforementioned 3D model. The fire model was, also, visualized and projected to the sand.

- **Pathfinding** : This phase includes the implementation of the pathfinding algorithms, in order to calculate the minimum-cost path to the fire, and the projection of this path on the tangible GIS.

- **Object Detection** : In this phase, we implement some object detection techniques to acquire information that can be combined with other parts of the thesis to give a more interactive aspect to the end product.

Once all the phases are completed, they are combined one by one and they are tested to observe whether they conflict with each other or not. At this point, we need to clarify that each of the phases consists of two sub-phases. In the first sub-phase, the implemented system is tested in a static environment, to study if the special mechanisms and algorithms function properly. Then, in the second sub-phase, the tests take place in a real time environment and we detect whether the approach we followed on the static environment test can function in real time or adjustments have to be made. The end product is implemented by combining all the already tested parts of our thesis. Then, we carry out more experiments to illustrate the correct function of the project as a whole.

5.1 Tangible GIS Visualization

The first experiments aimed to identify whether the basic part of the project was functioning correctly. Basically, what we wanted to observe is if the 3D model projection onto the sand was corresponding to the expected form of landscape. By expected form we mean that the projected landscape should function according to the thresholds that were defined in Section 4.3.3. Specifically, what we expect to observe is water particles at the parts of the sand where the height is low, grass at median height values and mountains at the parts with high elevation. We need to point out that ensuring the proper functionality of the visualization phase of our thesis, also, ensures that the height data acquired from the Kinect Sensor and the generated 3D model are appropriate too.

The experiments we carried out were relatively simple ones. They mainly consisted of altering the sand's form to study if the shape and appearance of the 3D model and thus the projection of the GIS would alter accordingly. In Figure 5.1 the results we get from different states of the sand's surface are shown, where someone can see the different landscapes that are projected on the sand as we mess around with its form.

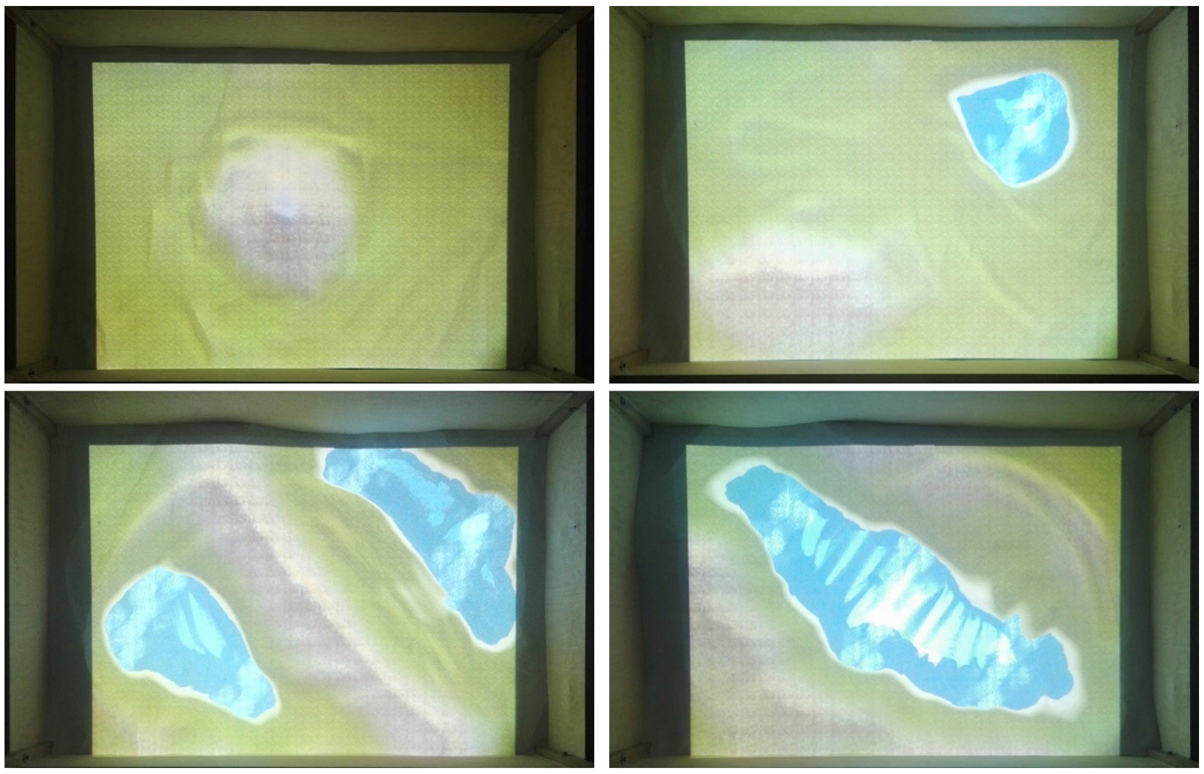


Figure 5.1: Different instances of the visualization of the sand's surface

5.2 Wildfire Modeling

Once the visualization phase of testing is completed and we ensure its proper functionality, we can proceed to the testing of the wildfire spread model. As mentioned multiple times is previous sections, the model of the fire is attached to the 3D model that is generated through the height data acquired from the Kinect Sensor. This means that various tests have to be carried out in order to ensure that the fire behaves according to the implemented fire model, as well as, that the fire model and the 3D model of the sand's surface collaborate properly.

The experimentation with the fire model can be separated into two parts. The first part focuses on confirming that every cell of the cellular automaton contains the correct information and that they update this information every time the form of the tangible GIS is altered. Then, we ensure that the fire is properly visualized, so that its position on the sand is clear to the observers. In the second part, we test whether the

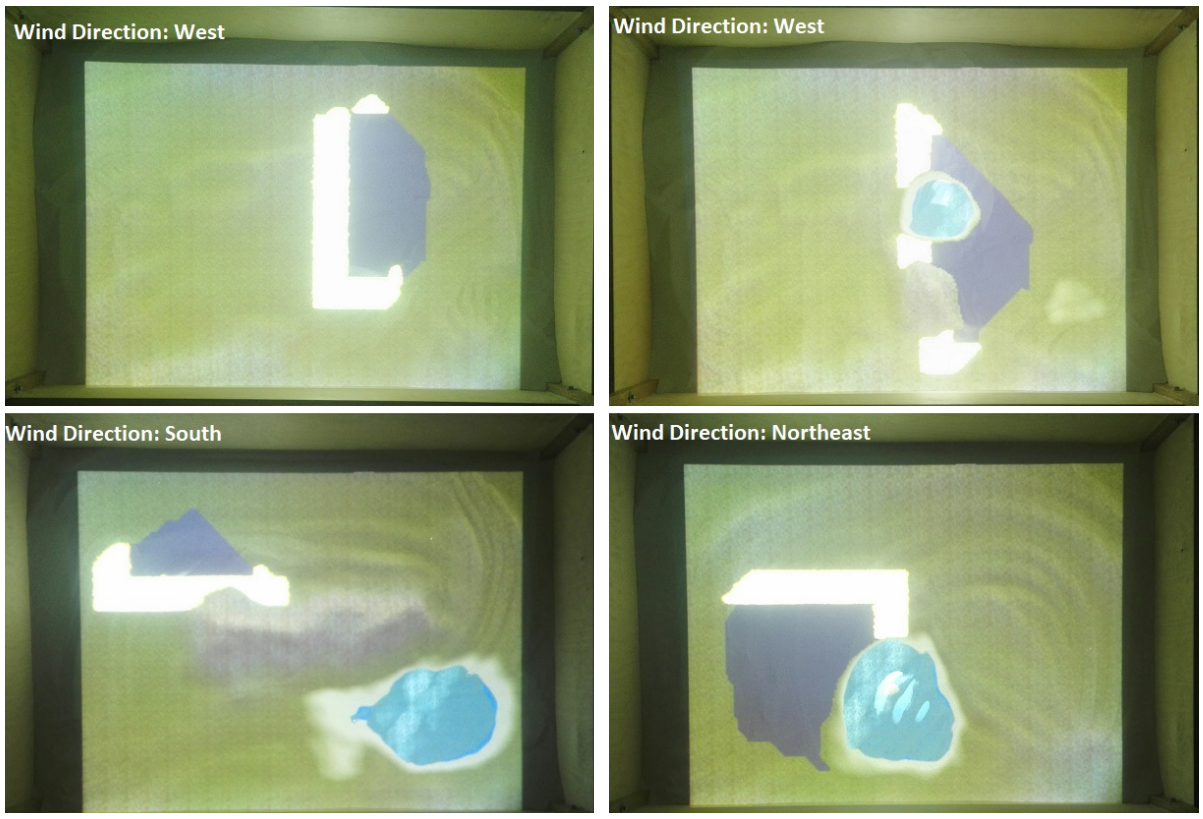


Figure 5.2: Different instances of the wildfire spread model

fire propagates accordingly in both static environmental conditions and altering ones.

In Figure 5.2 some instances of the fire's behavior in different weather and terrain conditions, as well as, its projection on the sand's surface can be seen.

5.3 Pathfinding

Completing the wildfire model testing enables us to test our implemented pathfinding algorithms. As mentioned in Section 4.5, we tested two algorithms, the A* search and the Value Iteration. However, we will only present the results we acquired by the Value Iteration algorithm, since A* was dropped out relatively early in our approach. The testing of the pathfinding went through many phases in order to ensure that it worked properly in all possible scenarios.

So, at first, we experimented with the simplest scenario: one agent and one single

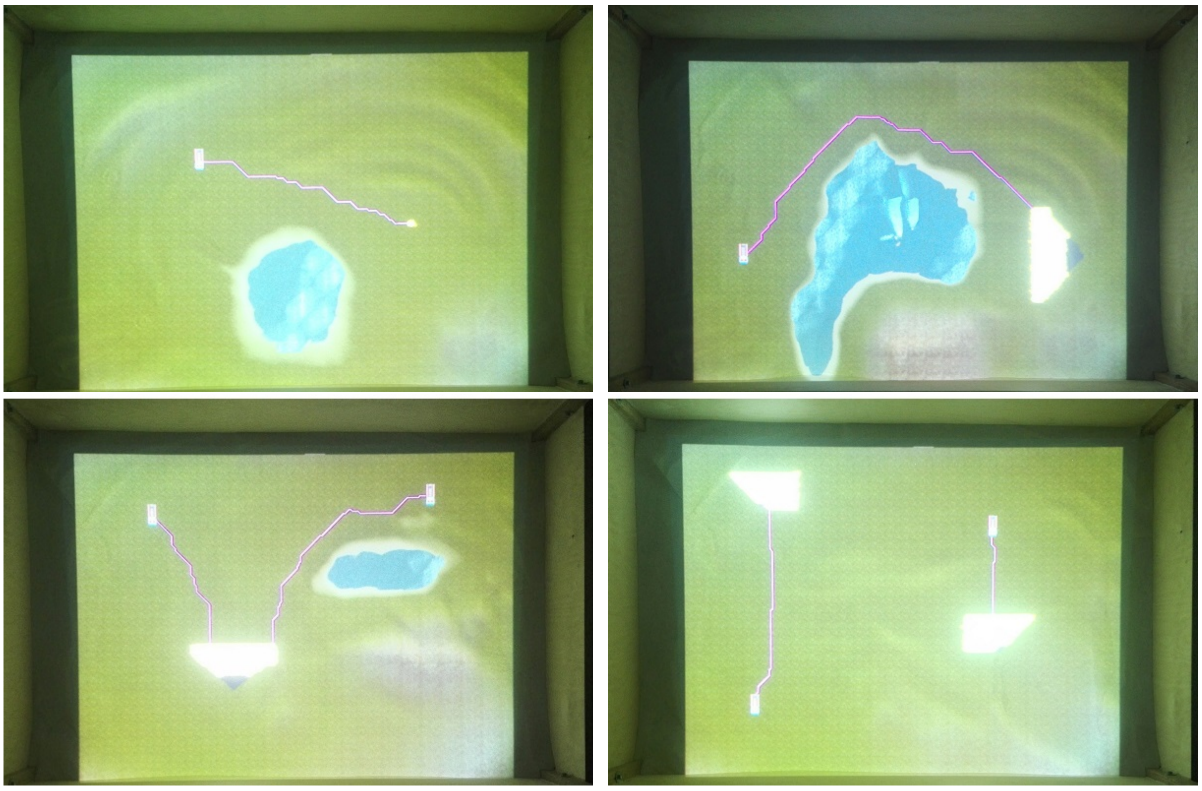


Figure 5.3: Different found paths in different conditions

fire cell as a target. What we wanted to observe was whether the path was the correct one in both cost and usability aspect. This means that the path had to be the minimum cost one and not pass from unwalkable parts of the map. Then, we experimented with multiple targets in both the forms of a bigger fire front and of multiple fire fronts, as well as, with multiple agents aiming to reach the same or different targets. Lastly, we tested out the algorithm in a real time scenario, to ensure that the output paths were updating correctly.

In Figure 5.3 the results of each phase described above can be seen, while in Figure 5.4 the real time recalculation of the path, when the terrain conditions are changes, is highlighted.

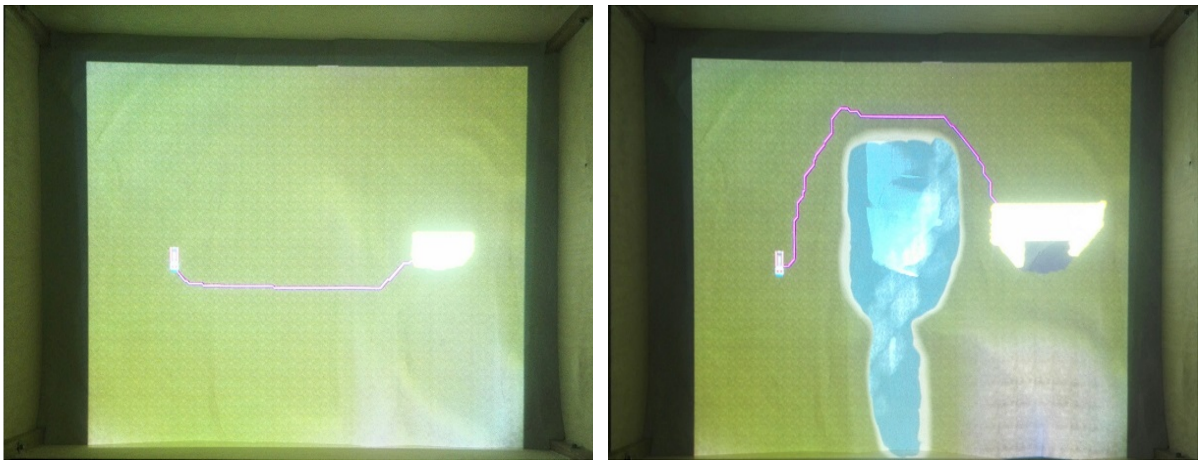


Figure 5.4: Real time altering terrain example

5.4 Object Detection

In this thesis, object detection was basically used as an assisting tool, in order to acquire information regarding some key variables of the simulation. However, that doesn't mean that the techniques that were used, should not go through testing, to ensure that they function properly. So, once all the simulation phases were completed, we run some tests to examine whether our object detection implementation was producing the correct data that we needed. Particularly, these data had to do with the position of specifically shaped objects on the sand and the correlation of the positions of the two circles that consisted the anemometer. The accuracy of this procedure laid in how close the detected positions were to the actual positions of the objects, as well as, how accurately the positions of the two circles were translated to the wind's direction and speed.

In the following two Figures, the results of the application of the object detection techniques on our thesis are shown. Specifically, in Figure 5.5, we showcase the detection of the objects on different positions on the sand, by highlighting them with black squares. Then, in Figure 5.6, we illustrate how the wind's direction and speed change as we move the pointer (smaller) circle around on the base (bigger) circle.

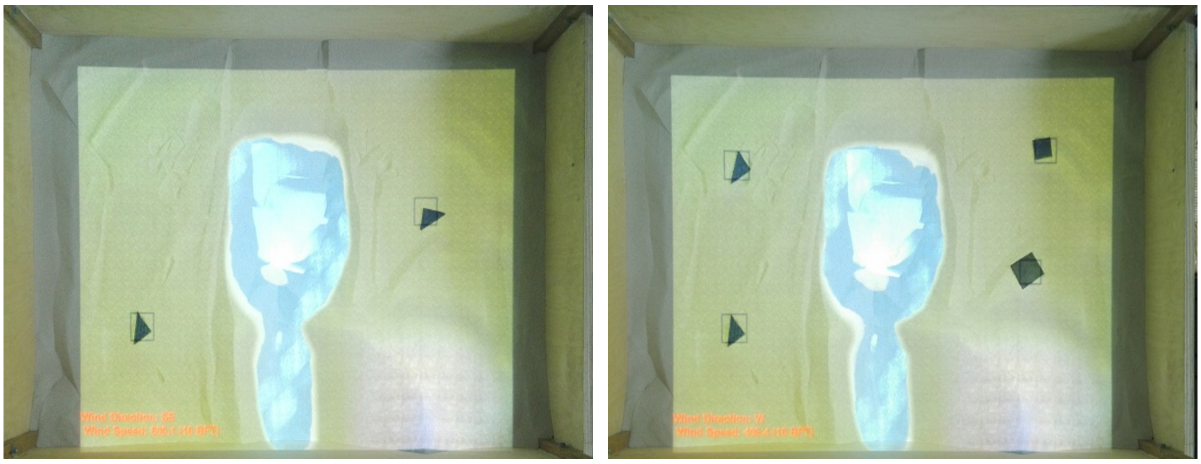


Figure 5.5: Examples of detected objects on the sand

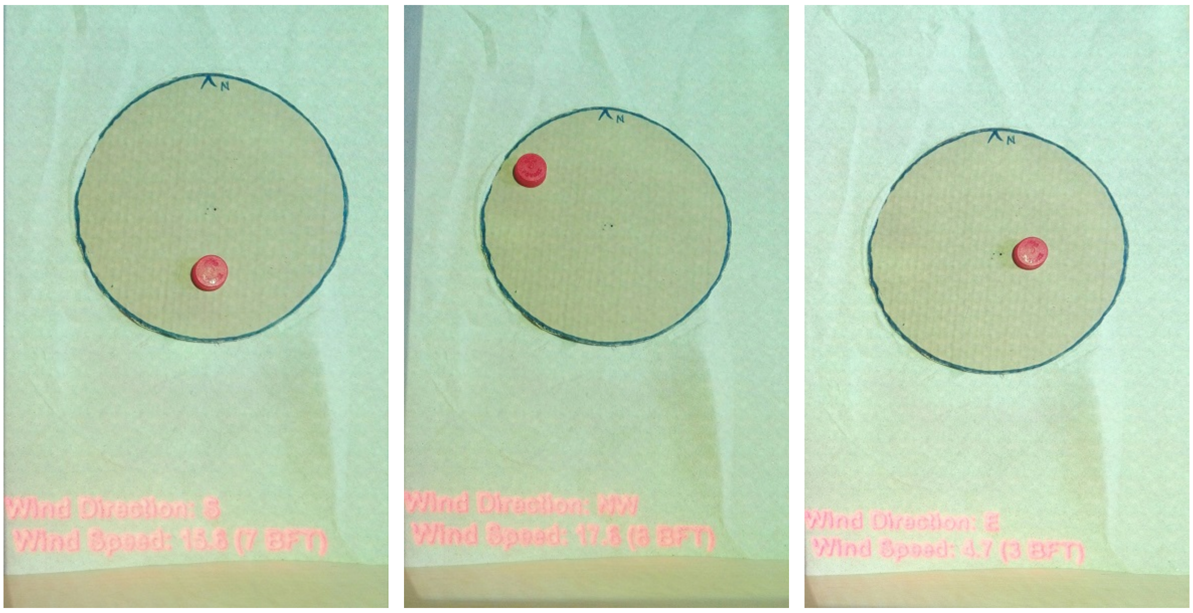


Figure 5.6: Some instances of the function of the anemometer

5.5 End Product

The final result and thus the end product of our thesis is the combination of the aforementioned tested parts. However, some final tests have to be carried out, in order

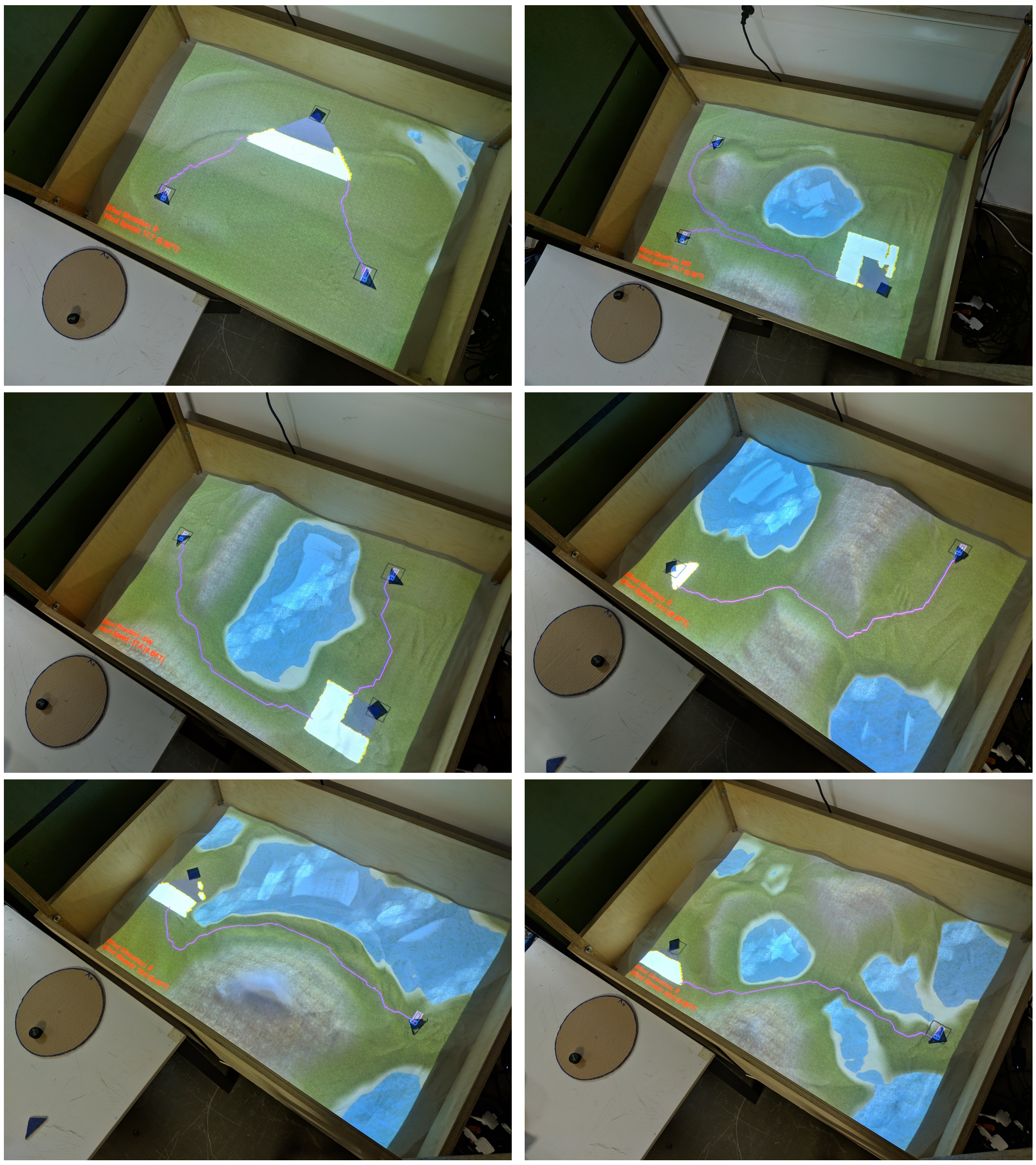


Figure 5.7: Different firefighting scenarios

to ensure that our project functions properly as a whole. Once, these final tests are completed the end product is a simulation of a firefighting scenario on a tangible GIS that can be altered in real time. Specifically, the surface of the sand is being scanned by the Kinect Sensor and the acquired height data are used to create a 3D model of it. Then, this 3D model is projected back on the sand. The users can interact with the sand and alter its form, which means that the 3D model and thus the projection of it on the sand with change as well in real time. Also, the users can place objects on the sand to determine the starting points of both the wildfire and the firefighters/agents, as well as, they can alter the wind conditions through the anemometer. Lastly, they can initiate the simulation of the firefighting scenario and interact with it in real time by altering the terrain or the wind parameters.

Some screenshots of various tested simulations can be see in Figure 5.7.

5.6 Comparison

As mentioned in Chapter 3, there are two approaches that are based on a tangible GIS in order to simulate physics-related dynamics or natural disasters. Specifically, the Augmented Reality Sandbox that was implemented in UC Davis [47] aims in illustrating both geographic and hydrologic concepts, while Simtable [11] aims in providing a training tool for disaster prevention teams, like firefighters.

Our approach has many similarities with the two aforementioned approaches in terms of both their setup and features. Particularly, all three of these approaches use sand to form the physical model of the tangible GIS and some sort of depth camera to digitize it, which in our case and UC Davis' case this camera is a Kinect Sensor. Also, every approach provides a real time visualization of the landscape that corresponds to the shape of the sand and the ability to dynamically alter the geological parameters of the GIS and thus its projection on the sand.

Apart from the common features, each of these approaches has some unique features too that no other approach has. Starting with UC Davis' approach, its unique feature is the simulation of hydrologic dynamics, meaning that the user can study the behavior of watersheds on different terrains. Also, the Augmented Reality Sandbox is very easy to use, making it ideal for use in schools or museums. However, this approach is limited to simulating only hydrologic concepts, which in turn does not make it an ideal tool in assisting in preventing or dealing with any form of natural disaster.

As for Simtable, its unique features lay in its advanced simulation model. Specifically, Simtable enables the user to define a vast amount of parameters that affect both the simulation of the natural disaster and the response to it. This means, that Simtable can be an excellent tool in assisting search and rescue teams and fire brigades, if used by someone with the appropriate experience and technical expertise. However, everyone who is lacking this expertise can find it really hard to use Simtable at its fullest or even use it at all. Also, the response that is projected throughout the simulation is a user driven one and does not update dynamically.

Our approach provides almost every feature that the other two approaches provide in a more simple and easy to understand and use way. The unique feature of our approach is the dynamic calculation and projection of the optimal response to the wildfire (our simulated natural disaster). This means that every time the parameters of the terrain alter, the minimum cost path is recalculated and reprojected on the sand's surface, in order to correspond to the new state of the tangible GIS. Last, but not least, the combination of the object detection techniques and the automatic calculation of the response make our approach a perfect tool for the training of firefighters that have no technical expertise.

Chapter 6

Conclusions

6.1 Conclusions

This thesis describes the implementation of an application for simulating firefighting scenarios using a tangible GIS in the form of a sandbox and the Kinect Sensor for Xbox One. The 3D modeling and visualization of the tangible GIS was implemented using the Unity3D game engine software. The simulation application takes as inputs both the color and depth image of the Kinect sensor's cameras and creates a polygon mesh based on just the depth image, on which a wildfire spread model and path-finding algorithms can run. The fire model created is grid-based and functions through a cellular automaton, while the minimum cost path, that the application suggests and highlights, is evaluated through the Value Iteration algorithm, which also functions on a grid created based on depth data obtained from the sensor. The color image is used to detect objects of various shapes placed on the sand or in some cases around it. These objects serve as the starting points of the fire or the anemometer used for the fire model that was created to simulate a wildfire spread, as well as starting points of the minimum cost path. Lastly, all the elements that comprise the simulation are projected in unison on the sand's surface via a single projector unit and the user can modify in real time most of the parameters that affect the simulation, like the shape of the sand's surface (depth data), the starting point of the fire or the wind's direction.

6.2 Future Work

6.2.1 Advanced Fire Model

Although the implemented fire model is a very good and realistically looking one, there are a lot more parameters that could be added to it. First of all, in this thesis the fire model lacks in terms of the diversity in both the vegetation's types and density, since only three categories were defined for each of them and in the end only some of them were actually used due to the limitations of the depth data. Various techniques can be used to improve these two parameters affecting the fire propagation, including the use of object detection or externally acquired data particularly acquired to be used as information for these two parameters. This will allow for a fire spread simulation that will resemble to reality even more than before and ,of course, a fire model that will be more useful to the firefighters that want to use the implemented application.

6.2.2 Points of Interest and Evacuation Plans

Apart from a more accurate fire spread model, another factor that could make the simulation of the firefighting scenario more helpful for the firefighting brigades would be the addition of points of different interest on the map. These points of interest could be cities located near the fire front or forests that need to saved. On firefighting scenarios points like these are of higher priority than, for example, points corresponding to rocky land or non fertile soil, where the fire cannot easily spread to, if not at all. Adding these points of high priority will force the minimum cost paths to guide the firefighters towards these points and provide a more accurate simulation of how the firefighting scenario should unfold in real life.

Furthermore, an equivalently helpful addition to the simulation would be that of an evacuation plan evaluation. Specifically, in case the fire front is headed or is near a city or a village, an evacuation plan could be evaluated by using the same algorithms used for the minimum cost path-finding, which the residents of that city or village could follow, in order to avoid any danger. This plan could be projected on the sand, so that it can be visually presented in case the simulation should be used for the education of both the firefighters and local people.

6.2.3 Additional Path-Finding Exploration

In this thesis, only two out of hundreds of algorithms used for finding a minimum cost path were tested. There are a lot more algorithms that can evaluate a path, not necessarily a minimum cost one, in various ways and with different speed. Also, the two algorithms that were used, in our case, can be further refined, so that they become more efficient and fast. So, through various experiments a better version of the used algorithms or even a brand new algorithm could be implemented to calculate the minimum path that the firefighters have to follow, in order to get to the fire in the least amount of time.

6.2.4 Simulation of Different Types of Natural Disasters

Lastly, this thesis focused only on the simulation of firefighting scenarios. However, there are plenty more types of natural disasters that can occur and can have the same, or even worse, catastrophic results. Fortunately, some of these disasters can be easily adopted to the implemented system by simply modeling their behavior. That is possible, because the visualization of the landscape was implemented through its 3D model, so the components of the disasters, like floods or earthquakes, could be applied directly on it. For example, the approach that was followed at the UC Davis project for the hydrologic aspects of it, could be used on the 3D model that was created with Unity3D to simulate rainfalls or floods.

References

[1] A* algorithm pathfinding. https://github.com/SebLague/Pathfinding. 71, 74

[2] Anatomy of a mesh in unity3d. https://docs.unity3d.com/Manual/AnatomyofaMesh.html. 13

[3] Canny edge detection. http://justin-liang.com/tutorials/canny/. 29

[4] Roger Tomlinson. https://web.archive.org/web/20151217012639/http://ucgis.org/ucgis-fellow/roger-tomlinson. 15

[5] depthjs open-source browser extension based on the kinect sensor. https://github.com/doug/depthjs. 7

[6] Emgucv. http://www.emgu.com/wiki/index.php/Main_Page. 86

[7] Introduction to polygon meshes. https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-polygon-mesh. 13

[8] Kinect for windows sdk beta. https://www.microsoft.com/en-us/research/project/kinect-for-windows-sdk-beta/. 7

[9] Official unity3d website. https://unity3d.com/. 9

[10] Procedural landmass generation. https://github.com/SebLague/Procedural-Landmass-Generation. 52

[11] Simtable. http://www.simtable.com/about/. 40, 43, 103

[12] Unity mesh smoothing. https://github.com/mattatz/unity-mesh-smoothing. 59

[13] NJGIN's information warehouse. https://njgin.state.nj.us/OGIS_IW/. 16

[14] Medical practice finds use for kinect hack. *QuickJump Gaming Network*, December 27, 2010. https://www.qj.net/medical-practice-finds-use-for-kinect-hack/. 8

[15] Doctors use xbox kinect in cancer surgery. *CHealth*, March 22, 2011. 8

[16] Alex Alexandridis, D Vakalis, Constantinos I Siettos, and George V Bafas. A cellular automata model for forest fire spread prediction: The case of the wildfire that swept through spetses island in 1990. *Applied Mathematics and Computation*, 204(1):191–201, 2008. 19, 62, 70

[17] Yali Amit and Pedro Felzenszwalb. Object detection. *Computer Vision: A Reference Guide*, pages 537–542, 2014. 28

[18] Bevan B Baker and Edward Thomas Copson. *The mathematical theory of Huygens' principle*, volume 329. American Mathematical Soc., 2003. 20

[19] Didier Bonvin. Connecting kinects for group surveillance. *Ecole Polytechnique Federale de Lausanne*, December 21, 2010. 7

[20] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. *Polygon mesh processing*. AK Peters/CRC Press, 2010. 13

[21] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008. 86

[22] John Canny. A computational approach to edge detection. In *Readings in computer vision*, pages 184–203. Elsevier, 1987. 28

[23] K.C. Clarke. Advances in geographic information systems, computers, environment and urban systems. volume 10, pages 175–184, 1986. 14

[24] Stephanie Crawford. How microsoft kinect works. https://electronics.howstuffworks.com/microsoft-kinect2.htm. 5

[25] Kinect Devs. Kinect 3d photo capture tool 0.7 released. http://www.kinectdevs.com/forums/kinect-photo-capture-tool-f18/kinect-photo-capture-tool-released-anaglyph-stereoscopic-vision-t8.html. 8

[26] Andrei Dumitrescu. Minnesota university team adapts kinect for medical use. *Softpedia*, March 15, 2011. 8

[27] Mark A Finney. Farsite: Fire area simulator-model development and evaluation. *Res. Pap. RMRS-RP-4, Revised 2004. Ogden, UT: US Department of Agriculture, Forest Service, Rocky Mountain Research Station. 47 p.*, 4, 1998. 20

[28] Mark A Finney et al. An overview of flammap fire modeling capabilities. In *Fuels management—how to measure success: conference proceedings*, pages 28–30. USDA Forest Service, Rocky Mountain Research Station, Fort Collins, CO, 2006. 20

[29] Wallace L Fons. Analysis of fire spread in light forest fuels. *Journal of Agricultural Research*, 72(3):93–121, 1946. 19

[30] P. Fu and J. Sun. *Web GIS: Principles and Applications*. ESRI Press, 2011. 16

[31] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*. Pearson Education, 2011. 32, 35

[32] Michael F. Goodchild. Twenty years of progress: Giscience in 2010. *Journal of Spatial Information Science (1)*, 2010. 14

[33] Bill Green. Canny edge detection tutorial. *Retrieved: March*, 6:2005, 2002. 29

[34] J. Illingworth and J. Kittler. A survey of the hough transform. *Computer Vision, Graphics, and Image Processing*, 44(1):87 – 116, 1988. 33

[35] Kostas Kalabokidis, Alan Ager, Mark Finney, Nikos Athanasis, Palaiologos Palaiologou, and Christos Vasilakos. Aegis: a wildfire prevention and management information system. *Natural Hazards & Earth System Sciences*, 16(3), 2016. 20

[36] Luke T Kelly, Andrew F Bennett, Michael F Clarke, and Michael A McCarthy. Optimal fire histories for biodiversity conservation. *Conservation Biology*, 29(2):473–481, 2015. 20

[37] Nicole Lee. Nasa's jpl maneuvers a robot arm with oculus rift and kinect 2, points to more immersive space missions. *engadget*, December 23, 2013. 8

[38] Jan Mandel, Mingshi Chen, Leopoldo P Franca, Craig Johns, Anatolii Puhalskii, Janice L Coen, Craig C Douglas, Robert Kremens, Anthony Vodacek, and Wei Zhao. A note on dynamic data driven wildfire modeling. In *International Conference on Computational Science*, pages 725–731. Springer, 2004. 19

[39] Farouk Messaoudi, Gwendal Simon, and Adlen Ksentini. Dissecting games engines: The case of unity3d. In *2015 International Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6. IEEE, 2015. 10

[40] Microsoft. Project natal 101. *Archived from the original on January 21, 2012*, June 1, 2009. 5

[41] Terrence O'brien. Microsoft's new kinect is official: larger field of view, hd camera, wake with voice. *engadget*, May 21, 2013. 7

[42] Constantine P Papageorgiou, Michael Oren, and Tomaso Poggio. A general framework for object detection. In *Sixth International Conference on Computer Vision (IEEE Cat. No. 98CH36271)*, pages 555–562. IEEE, 1998. 27

[43] Elsa Pastor, Luis Zárate, Eulalia Planas, and Josep Arnaldos. Mathematical models and calculation systems for the study of wildland fire behaviour. *Progress in Energy and Combustion Science*, 29(2):139–153, 2003. 20

[44] Anna Petrasova. Gis-based environmental modeling with tangible interaction and dynamic visualization. 17

[45] Ben Piper, Carlo Ratti, and Hiroshi Ishii. Illuminating clay: a 3-d tangible interface for landscape analysis. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 355–362. ACM, 2002. 18

[46] Emil Protalinski. Kinemote brings kinect to boxee and xbmc on windows. *TechSpot*, December 24, 2010. 8

[47] SE Reed, O Kreylos, S Hsi, LH Kellogg, G Schladow, MB Yikilmaz, H Segale, J Silverman, S Yalowitz, and E Sato. Shaping watersheds exhibit: An interactive, augmented reality sandbox for advancing earth science education. In *AGU Fall Meeting Abstracts*, 2014. 39, 43, 103

[48] Gwynfor D Richards. A general mathematical framework for modeling two-dimensional wildland fire spread. *International Journal of Wildland Fire*, 5(2):63–72, 1995. 19

[49] R.C. Rothermel. *A Mathematical Model for Predicting Fire Spread in Wildland Fuels*. USDA Forest Service research paper INT. Intermountain Forest & Range Experiment Station, Forest Service, U.S. Department of Agriculture, 1972. 19

[50] R.C. Rothermel, Intermountain Forest, and Utah) Range Experiment Station (Ogden. *How to predict the spread and intensity of forest and range fires*. General technical report INT. U.S. Dept. of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station, 1983. 19

[51] S.J. Russell, P. Norvig, and J.F. Canny. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall/Pearson Education, 2003. 21, 23

[52] Vald Savov. Kinect finally fulfills its minority report destiny (video). *engadget*, December 9, 2010. 7

[53] Maliene V., Grigonis V., Palevičius V., and Griffiths S. Geographic information system: Old principles with new capabilities. *GIS Technologies and Applications in Urban Design and Planning*, pages 1–6, 2011. 14

[54] J. Von Neumann and A.W. Burks. *Theory of self-reproducing automata*. University of Illinois Press, 1966. 20

[55] David R Weise and Gregory S Biging. Effects of wind velocity and slope on flame properties. *Canadian Journal of Forest Research*, 26(10):1849–1858, 1996. 68, 69

[56] Rasmus G Winther. Mapping kinds in gis and cartography. *Natural Kinds and Classification in Scientific Practice*, 2014. 17

[57] Jenna Wortham. With kinect controller, hackers take liberties. *The New York Times*, November 21, 2010. 7

[58] Peter Yap. Grid-based path-finding. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 44–55. Springer, 2002. 20

[59] Zhang Yongzhong, Z-D Feng, Han Tao, Wu Liyu, Li Kegong, and Duan Xin. Simulating wildfire spreading processes in a spatially heterogeneous landscapes using an improved cellular automaton model. In *IGARSS 2004. 2004 IEEE International Geoscience and Remote Sensing Symposium*, volume 5, pages 3371–3374. IEEE, 2004. 68

[60] Zhang Yongzhong, E Youhao, Han Tao, Zou Songbing, and Wang Jihe. A ca-based information system for surface fire spreading simulation. In *Proceedings. 2005 IEEE International Geoscience and Remote Sensing Symposium, 2005. IGARSS'05.*, volume 5, pages 3484–3487. IEEE, 2005. 20