

Exploring the possibilities of Extended Reality in the world of firefighting

Development of a firefighting simulator for the Belgian Navy

Janne HEIRMAN
Shivam SELLERI

Promotor: Prof. dr. ir. S. Desmet

Co-promotor: Ir. T. De Vleeschauwer
(Belgian Defence)

Masterproef ingediend tot het behalen van
de graad van master of Science in de
industriële wetenschappen: Elektronica-ICT
Optie: Internet computing

Academiejaar 2019 - 2020

©Copyright KU Leuven

Zonder voorafgaande schriftelijke toestemming van zowel de promotor(en) als de auteur(s) is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, kan u zich richten tot KU Leuven Campus Groep T Leuven, Andreas Vesaliusstraat 13, B-3000 Leuven, +32 16 30 10 30 of via e-mail iw.groept@kuleuven.be.

Voorafgaande schriftelijke toestemming van de promotor(en) is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

Acknowledgements

We would like to thank our supervisor prof. dr. ir. Stefaan Desmet and our co-supervisor Commander ir. Tom De Vleeschauwer for their help and guidance through this process. They were always ready to answer our questions and help us overcome the challenges along the way.

We would also like to thank the Belgian Defence but in particular the Belgian Navy and the Royal Military Academy for offering us this Master's thesis project. It was a learning experience like no other and it was a great opportunity to research a promising subject for the future of the Navy and for the world.

We would also like to show our gratitude towards Commander Michel Bellemans and all the personnel from the Damage Control Center for their firefighting expertise and their helpful feedback on all fire related matters. Without their help, this simulator would not have looked as realistic as it is. They were also the ones that provided us with the firehose nozzle which is an essential part of this thesis.

A special thanks also goes to Major Robby Haelterman from the Royal Military Academy and his colleague Charles Hamesse. Major Haelterman was the first person to offer us this thesis project in cooperation with the Navy. We are truly grateful for them to lend us all the material we needed for this thesis. They were always ready to help us find solutions for logistical problems that occurred and to give us ideas that have been incorporated in this thesis.

The One Bonsai team also deserves a big thank you for their help with the virtual firehose nozzle design. They modelled the design for us, which saved us a lot of time and work.

Finally, we would like to thank our families, for their mental support during the entirety of the thesis. They helped us out with numerous non-technical and logistical challenges. Their help and support proved to be quintessential during the COVID-19 lockdown. A special shout out goes to Janne's cat Snoes for the comfort she brought and to Shivam's dog Oby who sadly passed away during this thesis. He did not see the end of this project, but he was always there to help us out. He gave us the distractions we needed and a lot of love when things went sideways.

All by all, we have all the above-mentioned people to thank for the successful completion of this Master's thesis. They were the backbone of this project. Thank you all!

Abstract

Brandbestrijding is een essentieel opleidingsonderdeel voor de marine aangezien het de veiligheid aan boord moet garanderen. Maar deze opleiding is gevaarlijk, kostelijk en milieuonvriendelijk. Daarom is de marine op zoek naar een opleidingsvorm die de huidige opleiding kan aanvullen en minder risico's met zich mee brengt.

Het gebruik van een Extended Reality (XR) applicatie kan een oplossing bieden. De leerlingen kunnen zich zo volledig inleven in de situatie en geavanceerde technieken aanleren zonder dat ze in gevaar komen.

Deze thesis beschrijft de ontwikkeling, van design tot implementatie, van een Virtual Reality (VR) simulator en wordt uitgebreid met een proof of concept (PoC) van wat in de toekomst een volwaardige Mixed Reality (MR) simulator zou kunnen zijn.

Met de VR-simulator kunnen leerlingen in een volledig virtuele omgeving trainen. De instructeur kan steeds nieuwe scenario's creëren door op willekeurige locaties vuren te plaatsen in deze virtuele omgeving. Na de plaatsing van deze groeiende vuurhaarden, kan de leerling het virtuele vuur blussen met een draadloos spuitstuk dat uitgerust is met sensoren. Het originele spuitstuk in gebruik aan boord van de marineschepen, werd uitgebreid met 3D-geprinte stukken die de sensoren en microcontroller bevatten. De microcontroller stuurt de data van de sensoren over wifi naar een server die in de applicatie draait.

De MR-simulator laat de marine toe om in eender welke omgeving of compartiment van het schip te trainen zonder dat de ruimte gemodelleerd moet worden. Met behulp van een stereocamera ziet de leerling de echte wereld. Op de echte wereld als achtergrond plaatst men virtuele vuren, die met virtueel water geblust kunnen worden. De MR-simulator is een vereenvoudigde versie van de VR-simulator omdat deze veel meer processing power vraagt.

Deze thesis wordt afgesloten met een performance test. Voor beide simulatoren wordt het verbruik van de processor en de 'frame rate' onderzocht. Deze resultaten worden besproken en op het einde worden er nog enkele verbeteringen en uitbreidingen voor het systeem aangehaald.

Extended Abstract

Firefighting is a crucial part of the Navy's training program, as it must ensure the safety on board. This training is dangerous, expensive and environmentally unfriendly. That is the reason why the Navy is looking for a safer form of training that can enhance the current one. They are looking for a training tool that can prepare the trainees to tackle real fires.

Extended Reality (XR) technology could offer a solution. Using both Virtual Reality (VR) and Mixed Reality (MR) to create a virtual training environment that is tailored to their needs, could reduce the danger, limit their training costs and reduce the environmental pollution that is caused by firefighting training. Trainees can immerse themselves into a virtual world and train specific techniques without compromising the training experience.

This thesis describes the development, from design to implementation, of a Virtual Reality (VR) simulator, which is extended with a proof of concept (PoC) for a future, fully-fledged Mixed Reality (MR) simulator. Both simulators are developed in Unity and use state-of-the-art technologies, including the HTC VIVE Pro system and the ZED Mini stereo camera.

In the VR simulator, trainees are immersed in a fully virtual world. The instructor can create an endlessly number of scenarios by placing fires at different locations. After the fires are placed, they start to grow, and the trainee can start extinguishing the virtual fires using a wireless firehose nozzle controller. A standard issued Navy firehose nozzle has been modified to be used as a controller in the simulators. The original firehose nozzle is equipped with a set of sensors to track all its potential manipulations. 3D printed parts were designed and manufactured in house using a small 3D printer. The printed parts are attached to the nozzle and house the sensors and microcontroller used to track its manipulations. The microcontroller sends the data over Wi-Fi, using the User Datagram Protocol (UDP), from the sensors to the server that runs inside of the simulation application.

The MR application allows the Navy to train in any environment or compartment of the ship, without needing to model the room. Using a stereo camera attached to the headset, the trainee sees the real world around him/her. This image is overlaid with virtual fires. These fires can be extinguished with virtual water, using the same firehose nozzle as the VR simulator. The MR application is a simplified version of the VR simulator since it requires a lot more processing power to render the scene. Simplifications include no smoke particle collisions, no fire spreading and a more simplified fire growing algorithm. The beginning steps were made to include object detection in the MR simulator. If object detection, combined with the recognition of materials, is implemented, this would allow the fire to spread depending on a burning object's material.

This thesis concludes with a performance test. For both simulators, the processor usage as well as their frame rate are examined. The developed system is also compared to other existing systems from companies and other researchers. At the end, some future improvements, ideas and suggestions are listed to give an overview of how the system can be improved or extended to better suit the Belgian Navy's needs.

Keywords: Virtual Reality, Mixed Reality, Extended Reality, firefighting, Tangible User Interface

Contents

Acknowledgements	v
Abstract	vii
Extended Abstract	x
Contents	xiii
List of Figures	xv
List of Tables	xvii
List of Abbreviations	xix
1 Introduction	1
1.1 Nature and scope of the problem	3
1.2 Outline of the paper	4
1.3 Adaption of goals of this Master's thesis due to the Corona protective measures	4
2 Background Information	7
2.1 Definitions	7
2.1.1 Extended Reality	7
2.1.2 Virtual Reality	7
2.1.3 Mixed Reality	7
2.2 Analysis of existing systems	8
2.2.1 FLAIM Trainer	8
2.2.2 Fire Training with Unity and HTC VIVE	9
2.3 Related works	10
2.4 General background information	11
2.4.1 3D printing	11

2.4.2	Fire ignition	13
2.4.3	Fire spreading	14
2.4.4	Fire classes	14
3	Requirement Analysis	17
3.1	Use cases	17
3.2	Functional Requirements	18
3.3	Non-functional requirements	18
4	Design	19
4.1	Mechanical components	19
4.1.1	3D design	20
4.1.2	3D printing	25
4.2	UI components	26
4.2.1	HTC VIVE Pro	26
4.2.2	Firehose nozzle	28
4.2.3	VR Ready computer	32
4.3	Deployment diagram	33
4.4	Hardware components and schematics	33
4.4.1	Arduino	34
4.4.2	Sensors	35
4.5	Software design	38
4.5.1	VR simulator	38
4.5.2	MR simulator	39
5	Development and Implementation	43
5.1	Implementation environment	43
5.1.1	3D modelling and printing software	43
5.1.2	Unity	43
5.1.3	ZED Camera	44
5.1.4	Python and TensorFlow	45
5.2	Arduino and sensors	45
5.2.1	The board	45
5.2.2	The software	45
5.3	Implementation in Unity	47
5.3.1	General code	47

5.3.2	Implementation of the VR simulator	56
5.3.3	Implementation of the MR simulator	63
5.4	Object detection	65
5.4.1	Dataset	66
5.4.2	Model	66
5.4.3	Training	67
5.4.4	Object detection with ZED	68
6	Evaluation	69
6.1	Performance	69
6.1.1	VR simulator	69
6.1.2	MR simulator	70
6.2	Durability	71
7	Discussion	73
7.1	General considerations	73
7.2	Evaluation of the Navy's requirements	74
7.2.1	Evaluation of functional requirements	74
7.2.2	Evaluation of non-functional requirements	75
7.3	Comparison with related words	75
7.3.1	Comparison with the FLAIM Trainer	75
7.3.2	Comparison with the fire trainer by B. Schlager	77
7.4	General discussion	77
8	Future work	79
8.1	Improvements	79
8.2	Future technology	80
8.3	Suggestions and possible extensions	80
9	Conclusion	83
10	Bibliography	85
A	Calculations of the flow rate selector ranges	89

List of Figures

1.1	Firefighting training with a powder extinguisher.	1
1.2	Firefighting training with a firehose.	2
2.1	FLAIM Trainer demo at DCC.	9
2.2	From left to right: rendered mesh with radiators, emitting mesh and voxel grid. (Schlager, 2017)	10
2.3	Example of under-extrusion.	12
2.4	Clogged extruder nozzle (left) and normal extruder nozzle (right).	13
2.5	Fire triangle. (Hannah, 2019)	13
2.6	Fire classed and extinguishing agents. (Marsden Fire Safety, nd)	15
4.1	Technical drawing of the AWG TURBO 2230. (Le Couteur, O., personal commuin- cation, July 31, 2019)	20
4.2	Head.	21
4.3	Base of the measuring head.	21
4.4	Hollow tubes attached to the base with the wires of the LDR running through them.	22
4.5	Analogue sensor housing (black) with the Arduino housing (gold) on top.	22
4.6	Arduino housing.	23
4.7	Cover of the Arduino housing.	23
4.8	Cross section of the cover.	23
4.9	Tracker baseplate.	24
4.10	Backplate (gold) and insert (black).	24
4.11	Backplate with PCB (on top) and batteries.	25
4.12	Backplate with switch keeping the insert in place.	25
4.13	HTC VIVE Pro kit including headset, controllers and Base Stations 2.0. (VIVE, nda)	26
4.14	VIVE Tracker	27
4.15	Firehose nozzle with all components attached.	28
4.16	Three Hall switches with a magnet hanging above it.	29

4.17	Result of photogrammetry of the firehose nozzle.	31
4.18	3D model of the firehose nozzle.	32
4.19	HP Z VR Backpack computer. (Hewlett-Packard, nd)	33
4.20	Deployment diagram of the application.	34
4.21	Arduino MKR1000. (Arduino, nd)	34
4.22	Electrical circuit of the Hall sensor.	35
4.23	Hall sensor PCB design.	36
4.24	Electrical circuit of the Hall switch.	36
4.25	Hall switch PCB design.	37
4.26	Electrical circuit of the LDR.	37
4.27	Use case diagram of the VR simulator.	40
4.28	Use case diagram of the MR simulator.	41
5.1	ZED Mini (Stereolabs Inc., ndc)	44
5.2	Connection scheme of the Arduino	46
5.3	Structure of the payload of the UDP datagram	47
5.4	Main menu	55
5.5	Game view of the VR simulator	56
5.6	Fire with its <i>OverlapSphere</i> (green) in Unity	59
5.7	A fire with its possible spawn points (white) in Unity	61
5.8	MR simulator game view with laser pointer (blue) and plane detection (pink)	64
5.9	Object detection with a ZED camera (Stereolabs Inc., ndb)	68
6.1	Unity profiler during VR simulation with 15 small fires present	70
6.2	Unity profiler during VR simulation with more than 15 fires present	70
6.3	Unity profiler during MR simulation with no fire present	70
6.4	Unity profiler during MR simulation with one fire present	71
6.5	Broken backplate (early version)	72
6.6	Broken pins in Arduino housing (early version)	72
8.1	Firehose nozzle with indication of position for potential hole.	80

List of Tables

A.1	Measurements taken from the analog hall effect sensors (A1 and A2)	89
A.2	Average of both sensors for each position of the flow rate selector	90

List of Abbreviations

AABB	Axis-Aligned Bounding Box
AMOLED	Active Matrix Organic Light Emitting Diodes
AR	Augmented Reality
CPU	Central Processing Unit
DCC	Damage Control Center
DDR	Double Data Rate
FMD	Fused Deposition Modelling
FOV	Field of View
fps	Frames per second
GDDR	Graphics Double Data Rate
GPU	Graphical Processing Unit
GUI	Graphical User Interface
HMD	Head-mounted Display
IP	Internet Protocol
IPD	Interpupillary distance
LDR	Light Dependent Resistor
MR	Mixed Reality
PCIe	Peripheral Component Interconnect express
PLA	Polylactic acid
PoC	Proof of Concept
PTFE	Polytetrafluoroethylene
RBI	Reality Based Interaction
SCBA	Self-Contained Breathing Apparatus
SMD	Surface-mount Device
SSID	Service Set Identifier
STL	Stereolithography
TCP	Transmission Control Protocol
TUI	Tangible User Interface
UI	User Interface
UDP	User Datagram Protocol
VR	Virtual Reality
XR	Extended Reality

Chapter 1

Introduction

For the Navy, firefighting on board is essential for the safety of the ship's crew and for the execution of the operations at sea. Basic firefighting training of the crew consists of two parts. Firstly, theoretical courses cover the different classes of fires and their characteristics, in order to use the right type of extinguisher. Secondly, practical training is introduced to teach the trainees the techniques and manipulations needed to tackle a fire. Figure 1.1 and Figure 1.2 are two examples of the practical training.

Choosing the wrong fire extinguishing method can have fatal consequences. For that reason, trainees learn to manipulate different types of extinguishers. There is an emphasis on using the right technique and being able to manipulate the extinguisher blindfolded because in case of a real fire, the smoke build-up severely reduces visibility which means that looking at what you are doing becomes really hard or even impossible. Learning the right techniques and manipulations is thus a very important part of training.



Figure 1.1: Firefighting training with a powder extinguisher.

During the practical part, trainees will be immersed and exposed to hot fire training and its associated risks. Although instructors are always present to ensure the safety of all attendees, accidents

do happen from time to time. Even experienced crewmembers can make mistakes. This is the reason why most crewmembers have to follow an additional firefighting course before their deployment. This additional course is a rehearsal of all major firefighting methods learned and trains specificities before deployment. It guarantees that crewmembers get enough practise and are up to date with the latest safety measures when it comes to firefighting.

Introducing Extended Reality (XR) technology can offer solutions to both risk reduction and training of firefighting methods. On top, XR allows for the development of an endless number of training scenarios that cover the full ship. This includes scenarios that are too expensive or impossible to simulate in real life. The goal of XR is not to replace the hot fire training. These trainings will always be required as part of the firefighting education program. The goal of XR however is to introduce a step between the theory classes and the hot fire training. By introducing XR into the mix, more trainings can be organised which means crewmembers will be able to practise more and improve their skills in a safer and more cost-efficient way.



Figure 1.2: Firefighting training with a firehose.

As a result, this will reduce the risk of an accident during hot fire training as well in real intervention situations. Due to repetitive training in the virtual simulator, one obtains an increase of automation of operation in the real scenarios.

The purpose of this thesis is to explore the possibilities of XR in the firefighting scene and develop a working proof of concept (PoC) for the Belgian Navy. At the end, a working XR simulator might give a glance of what role XR technology can play for the Belgian Navy. The system should highlight the key points to consider when they develop their own firefighting simulator and it should demonstrate why they should embrace this new technology. All of this while providing a clear overview of both the potential as the limitations of XR.

During the course of this thesis, several engineering techniques and disciplines were combined to achieve the end result. This includes, knowledge about electronics to create the electronic circuits used to perform measurements, the ICT-skills to program the software in Arduino and Unity, 3D modelling and printing knowledge to create the parts needed to support the project and finally, machine learning to introduce object detection into the mix.

1.1 Nature and scope of the problem

During the practical part of training, real gas and fuel fires are simulated. This part of the training focusses on habituation of the trainee. Although the heat radiation in approaching a fire is one of the more dominant factors, trainees have to learn to get close to a fire and not freak out. They have to get used to the claustrophobic feeling once engulfed in smoke to the point you cannot even see your hands before your eyes. Using a self-contained breathing apparatus (SCBA) and a thermal imaging camera is also part of the skills trained. The most important part of the practical training is learning the techniques and procedures to call out a fire, approach a fire and then attack the fire. This part of the training is the most intense and asks a lot of focus and attention from every trainee because it is also the most dangerous one.

The use of an XR firefighting simulator can be used to prepare trainees prior to a real-life scenario. This way they are already familiar with the scenario and know what to expect when the real-life training begins. By making the trainees try the simulator first, instructors can already single out the trainees that are not ready for the real deal. This will improve the safety during a real training because the instructors will know on who to keep an eye on.

Virtual trainings also give more freedom to the instructors to create more challenging and complex situations for trainees. They could create simulations that are impossible or too expensive to recreate in real-life. Moreover, customized simulations by simulating ship specific scenarios helps trainees who will embark in the near future to get to know their shipping environment even better.

Hot fire trainings are necessary to ensure trainees are capable to handle real fires. However, they have significant economic and ecological drawbacks. Firstly, hot fire trainings are expensive because a lot of material is needed like, different types of extinguishers (CO_2 , powder, foam, ...), SCBA's, protective equipment (e.g. firefighting suits), hoses, nozzles and a lot of fuel. Secondly, the feasibility of hot fire trainings encounters more and more societal pressure. Polluting the environment is a present-day issue as burning diesel, gasoline and propane emits too much CO_2 .

An XR simulator can reduce the amount of pollution created per training session, because no ignition of fire is needed for virtual training sessions. It can also reduce the amount of materials needed to accomplish a similar amount of trainings which will bring the cost per training down. The calculation of the training costs of 2019 below gives an image of how expensive firefighting training really is. These numbers were provided by the Damage Control Center of the Navy. (Van Engeland, P., personal communication, January 23, 2020)

Amount of trainees: 1 153

For each trainee, at least one extinguisher of each kind is provided.

Cost of one CO_2 extinguisher and disposal: ± 100 euro

\Rightarrow Total cost of CO_2 extinguishers: $1\ 153 * 100 = 115\ 300$ euro

Cost of filling and examination of powder extinguisher: ± 57 euro

\Rightarrow Total cost of powder extinguishers: $1\ 153 * 57 = 65\ 721$ euro

⇒ Total cost extinguishers: $1\,153 + 65\,721 = 181\,021$ euro

For training purposes, the Navy uses fuels and gasses too. The amount used of each fuel and gas in 2019 is shown below.

Diesel: 37 209 l

Benzine: 1 760 l

Propane: 38 000 l

The calculations show that in 2019 the Navy spent at least € 181 021 on fire extinguishers only. This is without including the cost of the fuel and gas used during training.

Obviously, there are many benefits of using an XR simulator, nevertheless it is important to keep in mind that it will never be able to replace hot fire training. Although it reduces the amount of hot fire trainings, hot fire trainings will always be needed to offer an authentic training experience. The XR simulator merely means to augment the training and to create an intermediate step between theory and practise. An advantage of the XR simulator is that it can be used to train when conditions do not allow it (e.g. bad weather). It can also be used as an extension of the current training facility to allow more people to train at the same time.

1.2 Outline of the paper

This Master's thesis starts with a brief introduction of the goal, scope and nature of this project. It is followed by a chapter that gives background information that is needed to understand the subject matter that is discussed in this work. In the third chapter, the use cases and all requirements that the Navy imposed are discussed. Chapter four talks about the high-level design of the general system and each separate component is discussed. This goes from the mechanical components to hardware components and software. The fifth chapter, the development and implementation chapter, explains how each component of the system is implemented. Both the used tools and software, as the implementation of the sensors and simulators are discussed. In chapter six, the system is evaluated on multiple criteria. The results of this evaluation are discussed in chapter seven. The penultimate chapter is a future work chapter that discusses possible improvements and suggestions. Finally, this thesis ends with a conclusion about the work delivered during the course of this thesis.

1.3 Adaption of goals of this Master's thesis due to the Corona protective measures

Due to the coronavirus outbreak, the Belgian government took some protective measures. As a result, user tests could not be performed. However, this was an important aspect of this thesis, because this thesis is not just about researching the XR technology but also about building a Tangible User Interface (TUI) to train instructors and trainees of the Belgian Navy. This means

that user input is an important factor in the design process to create a user experience that is pleasant, intuitive and interactive. On top of that, the Belgian Navy already experienced issues with other VR simulators. All of this meant that this issue had to be addressed and a solution needed to be found. The way of working was altered, and short videos of the gameplay were sent to the Navy during the design and development process. This ensured getting some kind of feedback from them and being able to adapt the system to their needs and requirements. The main goal of this thesis did not change because of this altered way of working.

The original user tests would assess the user friendliness of the system. Is the system easy to setup and how long does it take an instructor to set it up? The tests would also assess the physics of the simulator. Does the interaction between fire and water feel realistic? Is the needed amount of water to extinguish a fire of a certain size correct? Does the spreading of fire between objects and inside objects feels realistic? Is the shape of the beam type correct? Is the smoke development limiting visibility like in real-life? The general gameplay of the simulator would also be evaluated.

Chapter 2

Background Information

To understand the full scope of this thesis and to be able to reproduce all the steps and results, one needs some background information. The background information varies from the basics of 3D printing to the study of existing VR firefighting systems.

2.1 Definitions

2.1.1 Extended Reality

XR is an umbrella term that covers Virtual Reality (VR), Augmented Reality (AR), Mixed Reality (MR) and every other immersive technology that combines the real and virtual world. The real world is extended with virtual artefacts. (Qualcomm, nd)

2.1.2 Virtual Reality

VR is a technology that creates immersive illusions using computer simulations in which one is teleported into an artificial environment with which he can interact. (Merriam-Webster, ndb) VR is a technology that creates immersive illusions using computer simulations in which one is teleported into an artificial environment with which he can interact.

The idea of immersing someone in an artificial environment dates back to the 19th century but it only recently gained traction due to the commercialization of VR-headsets by companies like Oculus and HTC. These companies opened up the possibility of experiencing VR to the broad public. They brought the technology to the forefront by introducing products like the Oculus Rift and the HTC VIVE. (Rubin and Grey, 2020)

2.1.3 Mixed Reality

MR sits right in between the physical world and the virtual world. Virtual objects overlay the real world. In contrast to AR, MR takes into account environmental input. This allows the virtual object

to interact with the real world, which gives a sense that these virtual objects are part of the real world. Examples of environmental input are plane detection, object detection and head tracking. (Microsoft, 2018) environmental input are plane detection, object detection and head tracking. (Microsoft, 2018) To achieve this, stereo cameras, in combination with head-mounted displays (HMDs), are used. A stereo camera is a camera that has two lenses. The distance between these lenses is comparable to the distance between the human eyes. (Merriam-Webster, nda) Because of the distance between them, both cameras have a slightly different view. By comparing the difference between the two images, the camera senses depth. This is similar to the way the human eyes sense depth and 3D movements. (Stereolabs Inc., nda)

2.2 Analysis of existing systems

There have already been attempts to create immersive VR firefighting simulators in the past. Inspiration can be drawn from these existing systems and technologies to solve problems encountered during the design and implementation phases. Studying existing systems can help understand the design choices that were made during the course of this thesis. Although there have never been attempts to create an MR firefighting simulator many parallels can be drawn from VR.

2.2.1 FLAIM Trainer

The FLAIM trainer (Figure 2.1) is a firefighting simulator developed by a company called FLAIM Systems based in Australia. The system creates dangerous, difficult and challenging situations for firefighters in virtual environment. This way firefighters can train safely while being confronted by life-threatening conditions. They developed the system together with the firefighting industry to develop a range of training scenarios from car fires, to plane fires and structural fires. FLAIM system highlights the realism of their visuals and the interactions between fire particles and different extinguishing agents. The FLAIM trainer also contains haptic feedback in the form of a heat vest that simulates the heat of a fire and a hose-line system that simulates the force exerted by a firehose. The system also has a breathing mask that can measure the oxygen consumed by the user. (Deakin Research, 2018)

FLAIM systems do emphasize that the FLAIM trainer is not a replacement for hot fire training but complements it. This is because the system can be used all year round and does not depend on weather conditions. It can also be used to train scenarios that are too dangerous and expensive to simulate in real-life. (FLAIM Systems, 2019)

Belgian Navy experts are evaluating a FLAIM trainer system since 2019. Their system (Figure 2.1) consists of a firehose nozzle (4), a hose-line system for force feedback (2), a personal heat vest for heat feedback (not visible on figure), an SCBA with built-in computer(1) and a breathing mask for oxygen consumption measurements (3). They test the system for the same reason as why they created this thesis. They wanted to have a system with which they could train their crewmembers in a safe environment using a more immersive technology than a computer screen. In their experience the FLAIM trainer was lacking this feature amongst others. One of the biggest



Figure 2.1: FLAIM Trainer demo at DCC.

deficiencies encountered with the FLAIM trainer is that it does not have customized scenarios that are typical to the maritime environment within which the Belgian Navy acts. The use of the virtual nozzle does not feel real nor does it match the model that is in use at the Belgian Navy. Therefore, trainees are not getting used to the procedures they will have to execute during hot training or during a real fire. Nor are they getting used to the essential manipulations they will have to execute on the firehose nozzle that is used by the Belgian Navy. This are the biggest reasons why firefighting instructors of the Navy are reluctant to use the FLAIM trainer as an educational tool. Currently the FLAIM trainer is used as a PR-tool to recruit new recruits for the Navy. It is also used as a tool to introduce instructors and students to the concept of VR training.

2.2.2 Fire Training with Unity and HTC VIVE

B. Schlager explained how she implemented a fire training simulator in her paper, *Building a Virtual Reality Fire Training with Unity and HTC VIVE*. The goal of her research was to create an immersive fire training software to train non-professionals in using a fire extinguisher. She used the Unity game engine and an HTC VIVE to create her software and emphasized that immersion was one of the major goals.

To achieve a higher immersive experience, she attached the controllers of the VR headset to real fire extinguishers. This way the trainee feels the real weight of the extinguisher which acts as a kind of haptic feedback.

In her game scene, she works with two types of objects, flammable objects and non-flammable objects. Flammable objects have a corresponding voxel grid shaped like the 3D model and accessed at runtime. Each voxel in the grid saves information such as current temperature, physical properties and current state. An internal software calculates a 2D raw image which loads the grid,

as can be seen in Figure 2.2 (right). Every flammable object in her scene consists of six radiators (Figure 2.2, left) that are responsible to transfer heat from one object to another.

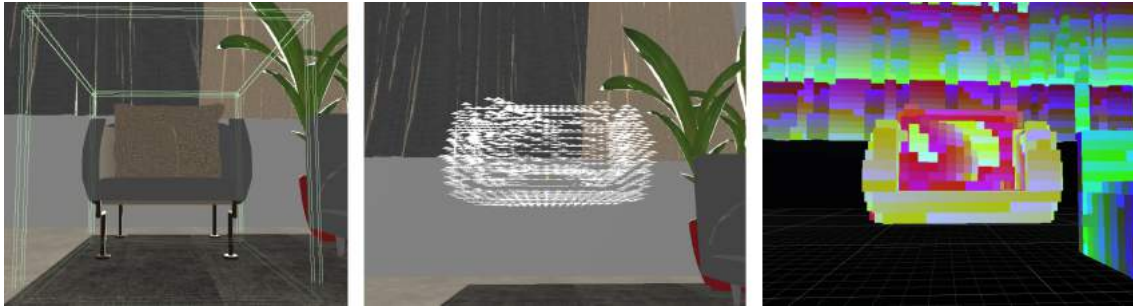


Figure 2.2: From left to right: rendered mesh with radiators, emitting mesh and voxel grid. (Schlager, 2017)

In her simulation, she was able to simulate heat transfer by, conduction, convection and radiation. She achieved this by using a heat transfer framework from her research institution to do the physical calculations based on the physical conditions of the environment and the material of the burning objects. She also relied on pre-processed data during runtime to mitigate performance issues and achieve a framerate around 90 fps.

Another feature she added was controls for the supervisor. The supervisor could interact with the environment and could control the fires. He is also able to see everything that the trainee does and is able to give live feedback. This way the training becomes a supervised learning process. (Schlager, 2017)

2.3 Related works

There have been several attempts to create an immersive fire training simulator by both companies and researchers. The design of each simulator achieves a specific goal. Some intend to train professionals, while others aim to train non-professionals and teach them the basic fire safety rules. Besides the different aimed audiences, each developed simulator runs for a different platform such as a desktop on one hand or a VR application on the other hand. After all, the common objective appears to be "education of people through gaming".

Learning through the act of playing serious games is a subject that has long been researched. But adding VR to the mix can have many benefits. VR is an immersive technology that can fully immerse someone in a simulated environment. This was not possible before and therefore it offers new possibilities to educate people. In his paper, *Immersive Interfaces for Engagement and Learning*, Chris Dede concluded that immersive virtual environments can enhance the learning process in at least three ways, allowing multiple perspectives, situated learning and transferring knowledge in the belonging context. (Dede, 2009) This means that using skills in a context that is similar to the real world can support the learning experience. If this is applied to the current context of firefighting, this means that if trainees are trained in an VR environment to fight fires like they would in real life, the immersive virtual environment could enhance their learning experience.

VR systems come with regular controllers that can be used for every type of application. This design choice is functional but takes away realism. Designing a specific controller for a specific application adds to both realism as to intuitiveness. This is how a Graphical User Interface (GUI) system can be distinguished from a TUI system. By designing a controller that is application specific, the designer can create user inputs that are based on real-world interactions. Basing interactions on pre-existing real-world knowledge and skills may reduce mental effort to operate a system. This can speed up learning or improve performance in stressful situations. This may also encourage improvisation and exploration because users do not need to learn interface-specific skills. (Jacob et al., 2008) Using a VR or MR firefighting simulator offers a more secure, practical and cost-effective alternative to traditional training while offering a more engaging experience. It also gives the freedom to train in a variety of training scenarios. (Schlager, 2017) These benefits make XR a worthy investment for the future.

2.4 General background information

This section contains general information about 3D printing, fire physics and a basic explanation of fire classes. It is important to have a basic understanding of these concepts to be able to follow the design choices that are made in the following chapters.

2.4.1 3D printing

After designing an object, it can be build using 3D printing technology. To be able to print an object via a 3D printer, the 3D model has to be transformed into instructions that can be understood by the 3D printer. This process is called slicing and can be done by a computer software called a slicer. The slicer will take an STL file that is exported from the 3D modelling software and will create a G-code file. This file contains all the necessary printer settings (e.g. Nozzle temperature, bed temperature, ...) and a whole set of instructions that will have to be executed by the printer. These instructions will make the printhead and print bed move to print the 3D design that was sliced.

3D printers allow to print custom 3D objects relatively easily. There are many different types of 3D printers that work in a variety of ways and are meant for different use cases. To make the 3D printed parts for this system a Fused Deposition Modelling (FDM) printer was used. FDM printers work by melting a thermoplastic material using a hot end and then forcing it out of an extrusion nozzle that is moving over a build plate. The extruded molten plastic is laid layer by layer on the print bed. Every time a layer is completed, the extruder is moved up to build the next layer on top of the previous one. By repeating this process till the last layer is completed, a full 3D object is printed.

The thermoplastic, also called filament, used in FDM printers looks like a long wire. Most often the wire is wound on a filament roll which is attached next to or on the 3D printer. Before the filament ends up on the build plate it has to go through a whole process. To start, the extruder pulls the filament from the roll. The extruder is made out of a spring guided lever and a stepper motor with an extruder gear on top of it. A spring mechanism pushes the filament against the extruder gear.

When the stepper motor rotates, the extruder gear rotates with it and pulls the filament in a certain direction. If the extruder is on the printhead, the filament is directly pushed through the extruder nozzle. If the extruder is a separate unit, a PTFE tube guides the filament to the printhead. Inside the printhead a heater heats the filament to the desired nozzle temperature set by the user. If set correctly, this temperature is equal or higher than the melting temperature of the filament. Once the filament melts it ends up on the print bed because the extruder is still pushing the filament into the printhead and through the extruder nozzle.

The whole process to go from a filament to a 3D object sounds simple and it is. Although, it has many possible failure points. The most common problems that happen during 3D printing are bad print bed adhesion, under-extrusion, over-extrusion and clogging. There are many other problems, but these are the major ones that can ruin a whole print.

Bad bed adhesion means that the print does not stick well to the bed. This can be due to a dirty bed, an unlevelled bed or an incorrect bed and nozzle temperature. This can be a major problem because it can cause the print to shift mid printing, which will ruin the whole print. It can also make the bottom layers of a part deform which is not a desired outcome and can render your part unusable. (Goldschmidt, 2019)



Figure 2.3: Example of under-extrusion.

Under-extrusion (Figure 2.3) and over-extrusion happens when the extruder is not extruding properly. This means that it is not feeding the filament at a consistent rate. When the extruder is not feeding enough filament to the extruder nozzle it can result in gaps, holes in the layers, thin layers or even missing layers. This is called under-extrusion and will compromise both the print quality as well as the print strength. Over-extrusion occurs when too much filament is fed to the extruder nozzle. This will result in blobs, stringing, dimensional inaccuracies and can even result in jams or clogging. (Hullette, 2019)

Clogging happens when the extruder nozzle gets clogged up (Figure 2.4). This can be caused by over-extrusion, incorrect printing temperature, a dirty nozzle, incorrect nozzle height, bad filament quality. There are many things that can cause clogging, so one should take care and clean his/her

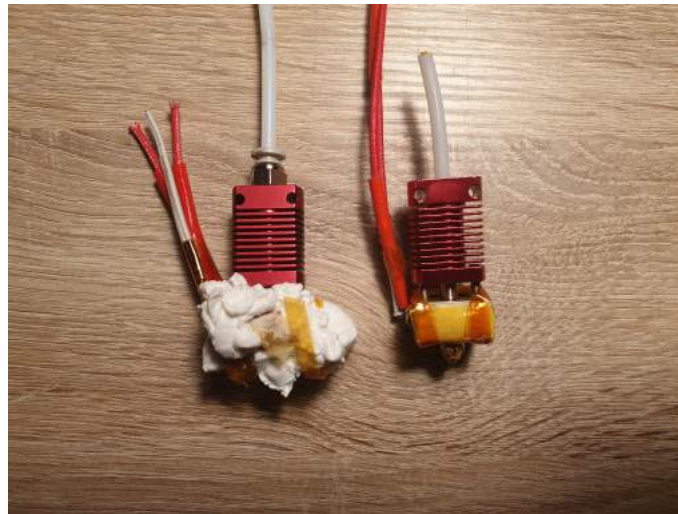


Figure 2.4: Clogged extruder nozzle (left) and normal extruder nozzle (right).

printer from time to time. Checking the printer settings and specially the temperature settings will help. One should also always use good quality filament. (Flynt, 2019)

2.4.2 Fire ignition

A fire is an oxidation process that combines oxygen with another substance at a very fast rate. Flames are the result of the combustion that happens when the energy created as part of the oxidation process is created faster than it can be dissipated.



Figure 2.5: Fire triangle. (Hannah, 2019)

There are three necessary elements to start a fire, namely heat, oxygen and fuel. These three elements form the fire triangle (Figure 2.5). As long as these three elements are present, the oxidation process will continue and as a result the fire will keep burning. (Hannah, 2019)

If one of the elements mentioned in Figure 2.5 is missing, no fire can be ignited. This also means that one could extinguish a fire by taking away one of the three elements. This is the exact goal of a fire extinguishing agent. An extinguishing agent will try to remove one of the three legs of the fire triangle to extinguish the fire. For example, water will take away the heat and thus the energy to

keep the oxidation process going.

2.4.3 Fire spreading

Fires spread by transferring heat energy to objects in their surroundings. This can be done in three possible ways, radiation, convection and conduction.

Thermal radiation is the emission of energy through electromagnetic waves. These waves travel in a straight line at the speed of light. An example of thermal radiation is the sun that heats the earth. This type of heating occurs between objects that are not touching. This is for example the way that a fire can jump from one object to another. Calculating the thermal radiation that is emitted by an object like a fire, is fairly complex.

Convection is the transfer of heat by the vertical movement of hot combustion gases. These hot combustion gases expand, become lighter than the surrounding air, and begin to rise because of upward pressure due to the decreasing density. This decreasing density creates a vertical upward stream. This pushes up heat and toxic black smoke to the ceiling. The denser layers of gas remain at the floor level. At this level the gas mixture contains more oxygen than burning gases. At the ceiling one will find the opposite mixing ratio with more toxic burning gases. So, the least dangerous environment in the compartment is at the ground level. This is the reason why it is instructed to stay close to the ground in a compartment that is on fire. The various temperatures and densities launch a thermodynamic stream creating an inverted convection cone of a mixed gas composed of hot air and smoke above the fire.

Thermal conduction is the transfer of heat within the material itself or between two touching materials. The conduction of heat inside an object depends on its material composition. Metal objects will be good conductors, while for example wooden objects are bad conductors and thus good insulators. Good conductors transmit heat very fast, while bad conductors transfer heat slowly. When talking about fire spreading, conduction is not an important factor to consider. (Auburn, nd) Except when metals are involved because metals will absorb the heat of the fire and will transfer it through the molecules of the material to other objects that are in direct contact with it. This means that any combustible material that is in contact with a heating metal can ignite through thermal conduction if their ignition temperature is achieved. (Hannah, 2019)

2.4.4 Fire classes

Different types of fires can be distinguished depending on the material of the burning object. These types are divided into five classes ranging from A to F. It is important for a firefighter to be able to distinguish the class of a fire because based on that information a different type of extinguishing agent must be chosen. Choosing the wrong extinguishing agent can have lethal consequences. Figure 2.6 gives an overview of the different classes and the extinguishing agents to be used.







						
	Wood, paper, plastics, soft furnishings etc.	Flammable liquids, petrol, oil etc.	Flammable gases, propane, butane, methane etc.	Metals; aluminium, magnesium, titanium, swarf etc.	Electrical apparatus, computers, phone chargers etc.	Cooking oils & fats
AFF FOAM	✓	✓				
WATER	✓					
WET CHEMICAL	✓					✓
CO2		✓			✓	
DRY POWDER	✓	✓	✓		✓	

Figure 2.6: Fire classed and extinguishing agents. (Marsden Fire Safety, nd)

Chapter 3

Requirement Analysis

After several meetings with the Belgian Navy, a requirements analysis was done to determine the use case, the functional and the non-functional requirements of the system to develop. This allowed the development of a system that best suits the users' needs and thus encompasses all their desired features.

3.1 Use cases

The Belgian Navy requires a VR/MR system to give firefighting training to its crewmembers while limiting the risk of injury and improving the flexibility to train in different environments. The system will create simulated fires in a classroom or on a ship for the benefit of the crewmembers.

The system will not serve as a replacement for the actual firefighting but will serve as an intermediate step between the theory and the actual firefighting training. This intermediate step will serve as a warmup for new trainees before they tackle real fires. The goal is to make them familiar with all the procedures they will have to follow in case of a real fire but doing so in a safe environment.

VR and MR are immersive technologies that make players forget about their real surroundings and immerse them in an artificial environment. Some players will not be able to distinguish between reality and the virtual situation they are immersed in. This will cause them to react in a way that is comparable to their reaction in a real situation.

The system will also be used with experienced crewmembers who have completed the fire safety training multiple times. They will use the system differently than new trainees will do. New trainees use the system to get used to the basics of firefighting, while the veterans will use the system to rehearse their firefighting skills and train specificities. They will use the system right before a deployment or during a ships deployment. This way they would not have to imagine fires during onboard fire drills, but they can just simulate them in a compartment of the ship using VR/MR.

3.2 Functional Requirements

Before the start of this project, the Navy set up some requirements that had to be fulfilled. These requirements have to do with how the system works, how it should behave and what its capabilities should be. These requirements are labelled as functional requirements.

The first functional requirement states that the system should implement VR and MR technology to create an immersive environment to train crewmembers in the art of firefighting. The simulation must be convincing enough to train crewmembers without gimmicks and distractions. This requirement encompasses the purpose of this thesis which is, delivering a working VR firefighting simulator and explore the possibilities of new technologies like MR in the firefighting scene.

The second functional requirement of the Navy is that the whole system should be compatible with their current equipment. They want to be able to use certain props to enhance the immersion and realism of the simulation. These props can vary from firefighting suits to big metal doors with which a player can interact in MR. The most important aspect that has to be considered to fulfil this requirement is to ensure a player's freedom of movement. This implies that no cables should hinder a player's ability to move around in the environment or to manipulate certain props and controllers. The third functional requirement is that a firefighting instructor should be able to place multiple fires on different objects in the training scenario and that these fires can propagate on their own.

The fourth requirement follows out of the previous one. Namely, the fires that are placed by an instructor must be extinguishable by trainees using a certain type of controller. The type or nature of the controller was not specified except the fact that it should react and be manipulated in the same way as the firehose nozzle used during real firefighting. This means that the user of the controller must be able to select the shape of the water spray, the flow rate and turn the water on and off.

3.3 Non-functional requirements

To meet all of the Navy's needs, they also stated some non-functional requirements. These are requirements that state how the system should be. This touches aspects like mobility, extensibility, intuitiveness and ease of use. These are all non-functional, but still essential requirements to suffice the needs of the Navy.

The first non-functional requirement is that the system could be setup in a classroom or a compartment of a ship. As a consequence, the system must be easy to setup and be movable at a moment notice. The whole system must be mobile enough to be moved around and not be fixed or dependent on a certain location.

Another reason why the system should be easy to setup is because the Navy requires that people without a technical background should be able to setup the whole simulator. This means that extra attention should be put on ease of use which is the second non-functional requirement.

The Navy formulated a third non-functional requirement which states that the system must be extensible. This means that the system must be built in a way that it can be updated or upgraded by themselves or a third party. For example, if the Navy wants to add new features, scenarios or even new types of controllers (e.g. different types of fire extinguishers) it should be easy to do so.

Chapter 4

Design

When designing a system, a lot of care should be taken to create the best user experience possible while achieving all the technical needs. A good balance has to be found between functionalities and ease of use. This balance has to be found during the design process, so it can be executed during the implementation process and be reflected upon in the evaluation process.

4.1 Mechanical components

The AWG TURBO 2230 (Figure 4.1) is the model of firehose nozzle used by the Navy during hot fire training and during active duty. Trainees have to get accustomed to this particular model. They have to know its capabilities by heart and should be able to manipulate it blindfolded. During firefighting, there is no time to look at the nozzle and thick smoke might prohibit the determination of its state. Therefore, Navy crewmembers must develop the skills to do everything by touch. The best way to accomplish this is by training with the same model over and over again. Thus, when designing the controller for the fire simulator it was obvious to choose the AWG TURBO 2230.

The nozzle was taken and fitted with sensors to track all its potential manipulations. An Arduino microcontroller is used to relay all that information to the computer in real-time. There are three movable pieces that have to be tracked for manipulations. First, there is the beam type selector (Figure 4.1, nr. 3), which is positioned at the front of the nozzle. This indicates which type of beam the user wants.

Secondly, there is the flowrate selector (Figure 4.1, nr. 4) which is positioned right behind the beam type selector and sets the flow rate of the water. Thirdly, there is the lever (Figure 4.1, nr. 5) which determines if water can come through. It is of utmost importance that every manipulation is tracked in real time so that the user does not experience any delay during the simulation. Delays, or better known as lag, can take the user out of the immersion.

In order to house all sensors, 11 additional components have been designed and 3D printed. These 11 components can be divided into two main parts. The first part is attached to the front of the nozzle and contains the sensors, the Arduino microcontroller and a LiPo battery. The second part is the backplate and contains a switch, batteries and LEDs. This part is attached to the back of the



Figure 4.2: Head.

On the bottom left side of Figure 4.3, it can be seen that a small part of the base has been cut out. This is done to be able to print this small part separately. This small part is glued to the inside of the nozzle to ensure that the whole head does not rotate when one accidentally tries to turn it. The importance of prohibiting the head from rotating comes from the fact that a tracking device will be mounted on top of it. The tracking device's orientation must stay the same relative to the nozzle itself otherwise the tracking in 3D space will malfunction. A correct orientation of the head is also important to ensure the proper tracking of the nozzle's manipulations.



Figure 4.3: Base of the measuring head.

On top of the base there is the analogue sensor housing which has two cut-outs to house two analogue Hall effect sensors (Figure 4.5, nr. 1). It was designed to have enough clearance for the sensors to properly read the magnetic field emitted by the magnets without bumping into them nor the movable outer black ring of the nozzle. This part also has a hole to guide all the cables from the LDR and the Hall effect sensors to the Arduino housing where they are connected to the Arduino microcontroller (Figure 4.5, nr. 2).

The Arduino housing (Figure 4.6) is the outer most part of the head and is connected to the analogue sensor housing. This part consists of two components, a bowl to fit the Arduino and the battery, and a cover to protect the electronics inside. The bowl has several important features. It

has a hole to guide the cables from the analogue sensor housing to the Arduino microcontroller (Figure 4.6, nr. 1). It has a raised bed to attach the Arduino with a Velcro strip and to leave enough space underneath the Arduino to attach the cables. On the side of the bowl, there is a cut-out to connect a USB cable to the Arduino when the cover is closed (Figure 4.6, nr. 3). On the outside, underneath the bowl there are three cut-outs to house the Hall effect switches that are directed towards the mouth of the nozzle. The last important feature on the bowl is a hole that goes all the way through the analogue sensor housing and the base towards the centre of the nozzle (Figure 4.6, nr. 2). This hole serves to guide a screw all the way into the nozzle, to attach the head to it using the threaded hole marked in Figure 4.1, nr. 1. It is very important to tighten this screw firmly to prevent the head from detaching from the nozzle. This firmly tightened screw together with the small base piece that is glued to the inside of the nozzle, are the only things to prevent the head from rotating.



Figure 4.4: Hollow tubes attached to the base with the wires of the LDR running through them.

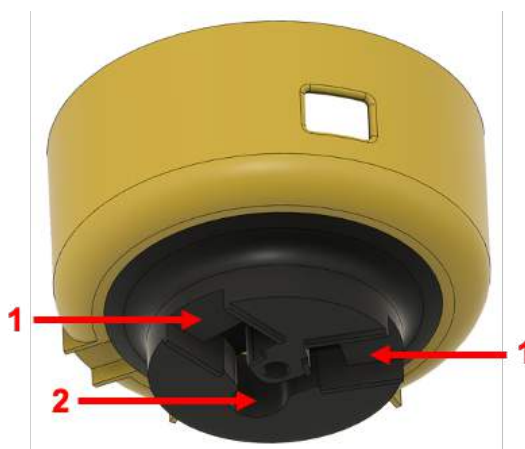


Figure 4.5: Analogue sensor housing (black) with the Arduino housing (gold) on top.

The cover of the Arduino (Figure 4.7) housing has three internal chambers that each house a magnet inside (Figure 4.8). The magnets are fully integrated into the print and are not visible from the outside. The only way to check their presence is to hold something magnetic in the neighbourhood of the cover. It was designed this way to ensure that no magnet would detach and go missing.

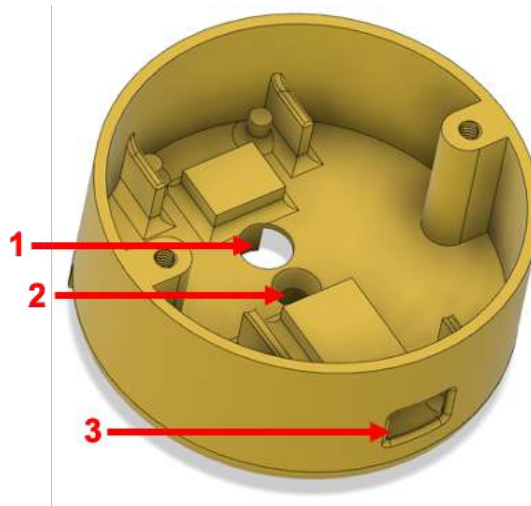


Figure 4.6: Arduino housing.



Figure 4.7: Cover of the Arduino housing.

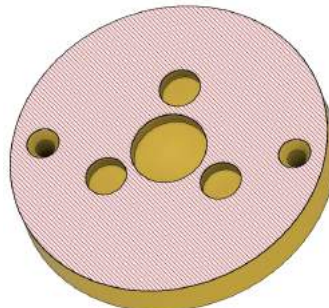


Figure 4.8: Cross section of the cover.

The magnets are used to secure the tracker baseplate (Figure 4.9). This is a simple plate that

holds a VIVE tracker by screwing it on using a 1/4" screw. The VIVE tracker is a tracking device that is used to track the movements of the nozzle in three-dimensional space. The plate also has three internal chambers which house magnets. These chambers are aligned with the chambers in the cover when the tracker is correctly attached to the head. The magnets were inserted into the cover and baseplate in a specific way that ensures that the baseplate can only be mounted in one specific orientation onto the cover. This ensures a correct oriented tracker at all time.



Figure 4.9: Tracker baseplate.

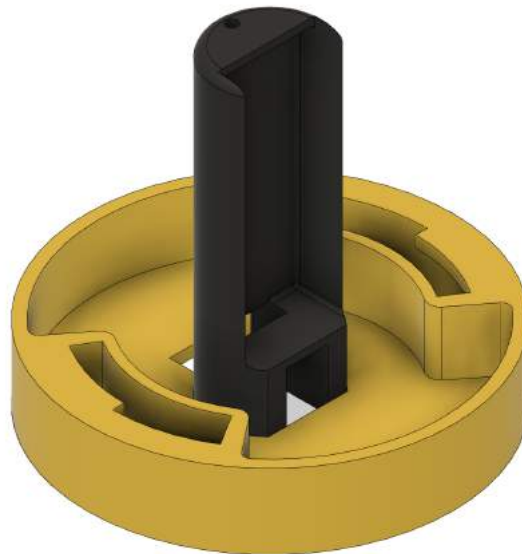


Figure 4.10: Backplate (gold) and insert (black).

The final 3D printed object is called the "backplate" and consists of two pieces (Figure 4.10). The first part is a battery holder which houses a battery pack for two AAA batteries. A small prototype board with three LED's is mounted on top of the battery holder (Figure 4.11). These are used when tracking the movement of the lever. The second part is the plate itself which is screwed on the back of the nozzle. This plate has a cut-out to slide in the battery holder. After sliding in the battery holder, it is held in place using a switch that fits right into the remaining gap (Figure 4.12).



Figure 4.11: Backplate with PCB (on top) and batteries.



Figure 4.12: Backplate with switch keeping the insert in place.

4.1.2 3D printing

After designing and slicing the 3D parts, a 3D printer with PLA filament is used to print each part. The parts are printed with black and gold PLA filament to match the colours of the firehose nozzle. PLA filament is currently the most basic and most used 3D printing filament. It has simple printing requirements that can be handled by the most low-end 3D printers. The used 3D printer is a very basic printer and the only printer available at the time. For that reason and the fact that PLA was the only filament at hand, PLA was chosen to print the 3D parts required.

4.2 UI components

The user interface (UI) is the main way a user is going to interact with the system and thus it plays a crucial role. It should be designed to be user friendly and it should encourage interaction with users. The UI of the firefighting simulator consists of three major components: the VR system, the firehose nozzle and a computer. Each UI component will be discussed more in depth to get a better understanding of the design choices made.

4.2.1 HTC VIVE Pro

The HTC VIVE Pro system is one of the best VR systems currently available. It won the award of best headset of the year on the VR Awards 2018. (VR Awards, 2018) The main parts of the system are the VIVE Pro headset, also called a head-mounted display, the VIVE Pro controllers, and the Base Stations 2.0. (Figure 4.13)

The headset is equipped with a dual AMOLED 3,5" screen with a resolution of 1440x1600 pixels per eye. It has a refresh rate of 90 Hz and a field of view (FOV) of 110°. The HMD is also equipped with a set of sensors that are used to determine the position and orientation of the headset. These sensors include (VIVE, ndb):

- SteamVR tracking
- G-sensor
- Gyroscope
- Proximity sensor
- Interpupillary distance (IPD) sensor



Figure 4.13: HTC VIVE Pro kit including headset, controllers and Base Stations 2.0. (VIVE, nda)

The default input devices of the system are the controllers, also called Wands. These devices track the movements of your hands and are used to interact with the virtual world. Each controller has multiple buttons, a trigger and a trackpad. A custom action can be mapped to each of these through the SteamVR bindings UI.



Figure 4.14: VIVE Tracker

The system can be extended with an HTC VIVE Tracker to track custom objects (Figure 4.14). When attached to a real-life object, these trackers track the movement of the object in space. On the back of the tracker, there are six pogo pins to which hardware can be attached. Pogo pins are spring-loaded electrical connectors to which wires can be connected. The force of the spring keeps the wire in place. The data from these hardware components can be sent directly to the VR game.

4.2.1.1 SteamVR Tracking 2.0

The Base Stations 2.0 alternately emit vertical and horizontal laser sweeps of infrared light over an angle of 120° . The photodiodes on the HMD's and controllers' surface pick up the signal. The position and orientation are determined by the difference in time at which the sensors were hit. This is called the inside-out tracking method. (Niehorster et al., 2017)

SteamVR Tracking 1.0 uses sync pulses in combination with sweeps. All sensors on the HMD, controllers and trackers sense the pulse at the same time. They start counting until they sense the vertical and horizontal laser sweep. The time between the pulse and the laser is used to determine the position of each sensor. Combining all sensors of a tracked object allows SteamVR to calculate the orientation, velocity and angular velocity. This is all done at a refresh rate of 1000 Hz. (Valve Corporation, nd)

SteamVR Tracking 2.0 is an upgrade of SteamVR Tracking 1.0 and does not use the infrared pulse anymore. Instead, data is encoded in the laser that sweeps through the room. This data is used to calculate the position, orientation, velocity and angular velocity. (Valve Corporation, 2017)

4.2.2 Firehose nozzle

Most of the controllers used with VR-headsets are all-purpose type of controllers. This means that they can be used in a variety of situations for a variety of tasks. The controls are just remapped for every situation. These controllers are not intuitive and have a low affordance. This is the reason why a real firehose nozzle is implemented as a controller (Figure 4.15). Using such a controller creates an interaction that feels like interacting with the real, non-digital world and that draws strength by building on a users' pre-existing knowledge. This type of interaction requires less mental effort from the user and improves their performance in stressful situations. (Jacob et al., 2008) Because the user is a firefighter in training and thus has prior knowledge about how the nozzle works and feels, and because the training scenario consists of extinguishing fires which is a highly stressful situation, using such a controller will have great benefits for both the user experience and the user's performance.



Figure 4.15: Firehose nozzle with all components attached.

4.2.2.1 Input tracking

The fire nozzle consists of three movable pieces that have to be tracked. The first one is the beam type selector that can rotate to three specific positions to select the shape of the water beam. Three hall-effect switches and a neodymium magnet (Figure 4.16, nr. 1) are used to track the movement of this selector. The neodymium magnet is glued to the black ring of the firehose nozzle. The three switches (Figure 4.16, nr. 2) are glued in their respective slots on the 3D printed head and are facing the magnet. The slots are at an angle respective to each other that is equal to the rotation angle of the beam shape selector to go from one shape to the other. This means that at each position

of the beam shape selector, the magnet will face only one switch. This will toggle the switch which will send out a signal to the Arduino.

The area where the magnet is placed is small, this required a magnet with a small footprint. But the magnet had to be strong too, because at the most extreme setting, the magnet is at 1,2 centimetres from the sensor. Neodymium magnets are very strong magnets, even small neodymium magnets have a large magnetic field that can bridge the gap of 1,2 centimetres. That is why they were chosen for the job.

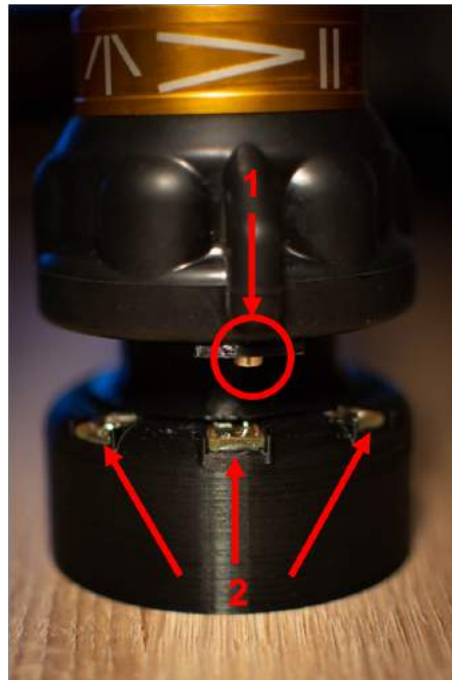


Figure 4.16: Three Hall switches with a magnet hanging above it.

The second movable piece that had to be tracked is the flow rate selector. This ring is in the middle of the nozzle and its rotation cannot be measured from the outside nor the inside. Sensors cannot be placed on the outside of the ring because the wires will have to run over pieces that will be manipulated by the user. This would hinder the user and the wires may be accidentally snapped or disconnected. Putting a sensor on the outside is thus a bad idea but putting a sensor on the inside is impossible because the firehose nozzle cannot be opened up. The only way to track the rotation of the middle ring is by tracking the movement of the whole front end of the firehose nozzle. The whole front part moves along the centre axis of the nozzle when the middle ring is rotated. The 3D printed head stays stationary because it is attached to the inside of the nozzle. This means that the translation of the front end of the nozzle can be measured relatively from the 3D printed head. To achieve this, two hall effects sensors and two regular magnets are used. The magnets are glued to the front end of the nozzle and the two hall effect sensors are placed in their respective slots on the 3D printed head. When the flow rate selector is rotated, the magnets move relative to the hall effect sensors and thus the hall effect sensors output a different voltage that is measured by the Arduino. The use of two hall effect sensors was done to have redundancy. The output of an analogue sensor

has a lot of jitter which makes the output voltage vary. By combining the output of two sensors and taking the average a more accurate result is obtained.

The third movable piece that has to be tracked is the lever. Tracking the lever by measuring its rotation was not possible because of the same reasons as the flow rate ring. There was no possibility to drill a hole in the nozzle because of its construction and material. To measure the lever another approach had to be taken. This approach consisted of an LDR and three white LED's. The LED's would be mounted on the back of the firehose nozzle and shine inside the nozzle towards the front. In the front an LDR is placed to measure the incoming light. Inside the nozzle there is a valve that opens and closes when the lever is rotated. Thus, by rotating the lever, the amount of light of the LED's that reaches the LDR varies. When its open, all the light can pass through and when its closed, no light passes through.

This LDR-LED-system is simple and circumvents all the constraints that the fire nozzle has to offer. Furthermore, does not hinder the user experience. The only downside is that the relation between the rotation and the resistance of the LDR is not linear. As a consequence, a way of mapping the rotation of the lever to the output of the LDR had to be found.

4.2.2.2 Wireless connection

The fire nozzle is used as a controller to play the game, which means that the player will be manipulating it the whole time. The player will be moving around in 3D space while using the nozzle. To not hinder the players movements and user experience, the controller is wireless. This implies that all the sensor data is transferred over a wireless network using a microcontroller. This way there are no wires or cables that can get tangled in equipment or props that the user is wearing to enhance his/her experience. This ensures that the second functional requirement of the Navy is fulfilled.

The wireless technology chosen to send data is Wi-Fi. This is because Wi-Fi is a universal technology that is adopted in most electronic devices like computers, laptops and tablets. Another option would be Bluetooth, but for Belgian Defence computers, it is more common to have a Wi-Fi module than a Bluetooth module. Wi-Fi also gives more freedom when the system is expanded in the future. This way the firehose can be programmed to communicate with multiple devices at once and it also gives the possibility to work with multiple devices at the same time. Wi-Fi has a larger range than Bluetooth which means that it can be used in a bigger environment. Currently, the system is designed for single player use in a classroom or a compartment of a ship. But if the Navy wants to expand the system to be used in a big hanger, with multiple players and devices, they can do so with much more ease using Wi-Fi than using Bluetooth. This satisfies the third non-functional requirement of the Navy.

To establish a Wi-Fi connection between the nozzle and the computer, a router is needed. This router will be configured with a certain Service Set Identifier (SSID) and password that will be pre-programmed in the microcontroller of the nozzle. This way the microcontroller can automatically connect to the router once it is turned on. The computer will have to be connected to the router as well and will receive a static IP address. This means that the computer will always get the

same IP address when it connects to the network. As a consequence, this IP address can be pre-programmed in the microcontroller to ensure it sends its UDP packets to the right server.

4.2.2.3 3D model

The physical firehose nozzle had to be represented by a 3D model in VR. This model is the main object a user will interact with during gameplay and thus it has to be a convincing representation of the actual firehose nozzle. There are several ways a computer model can be created of a real-life object.

First of all, there is photogrammetry which creates 3D models using overlapping photographs taken from objects, structures or environments. This is done by extracting 3D information stored in the combination of these photographs and reconstructing the surfaces. This technique is often implemented to create topographical maps, meshes, or point clouds based on the real-world. (Aber et al., 2019) Using this technique and the photogrammetry software Meshroom, a 3D model of the firehose nozzle was created. It took a lot of time to complete the process and the result was not satisfying (Figure 4.17). The model was incomplete, and the lighting used when taking the photographs is baked into the model. As a consequence, the nozzle did not feel like it belonged in the VR environment where it was supposed to be used. That is why another technique was required.

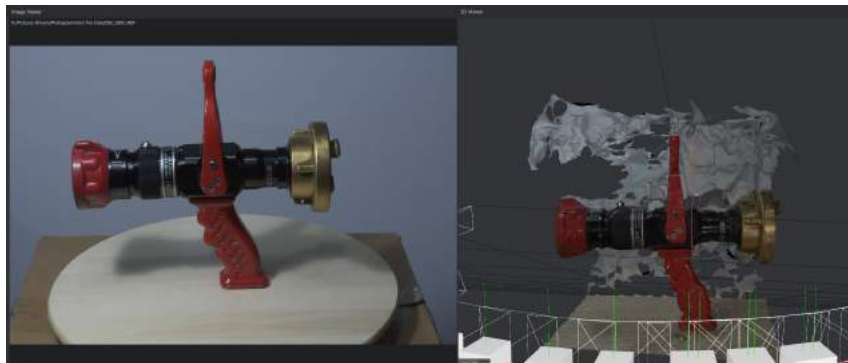


Figure 4.17: Result of photogrammetry of the firehose nozzle.

A second approach is a structured-light 3D scanner. This type of scanner emits light patterns onto an object. These light patterns get distorted by the object and are read by two cameras that are positioned on either side of the light source. The data collected by the two cameras is analysed to calculate the surface information and create a 3D mesh of the object. To create a mesh of the entire object, the object is rotated 360° and at each increment, the capturing process is repeated. This technique did not prove to be a success either, but that was due to the lack of essential parts of the scanning equipment used.

The last technique is to model the firehose using 3D modelling software like Blender. This is the technique that resulted in the final model in the simulator. This is due to the fact that photogrammetry and 3D scanning did not produce the desired result. Also, because One Bonsai, a company that is employed by the Belgian Defence, offered their 3D model of the firehose nozzle to be used in this project (Figure 4.18). The model needed to be textured and the origin of the different components



Figure 4.18: 3D model of the firehose nozzle.

had to be changed before being used in the simulation. The origins were altered because it made the rotation movement easier to implement in Unity.

4.2.3 VR Ready computer

In the XR application, the user needs to be able to move around freely. This means that cables between the user and a stationary computer need to be avoided. There are two ways to achieve this: a wireless adapter for the HMD or a backpack computer.

In the default setup, the transmission of data between the headset and the computer is sent through a cable. With the wireless adapter, the cable is no longer needed. The VIVE Wireless Adapter is a device that can be mounted on the headset, so it sits at the back of the user's head. In order to be able to use the adapter, some modifications to the setup need to be made. There are also some additional requirements. The adapter can only be used with a desktop computer with an empty PCIe slot. (VIVE, ndc)

The other option is to use a backpack computer. A backpack computer is a wearable PC that can be mounted on a harness and is battery powered. It can also be used as a normal desktop computer. The cables from the HMD go directly to the computer on the back of the user and not to a stationary computer. For this project, the Navy provided us with such a computer.

The backpack computer used for this project is the HP Z VR Backpack G1 (Figure 4.19). This computer is equipped with an Intel Core i7-7820HQ processor. This four-core processor has a clock speed of 2.9 GHz and can go up to 3.9 GHz with Intel's Turbo Boost Technology. The computer has Intel HD Graphics 630, but this integrated graphics card is disabled on this computer. The dedicated graphics card in this computer is an NVIDIA Quadro P5200 with 16GB of GDDR5. The PC is equipped with 32 GB of DDR4 RAM. The computer is running Windows 10 Pro 64. (Hewlett-Packard, nd)

VR applications demand a lot of processing power. This power is mostly needed for rendering, but also for physics simulations and other functionalities. This is why a computer with a good CPU and GPU is required to run such an application. The HP Z VR Backpack is a VR Ready computer and



Figure 4.19: HP Z VR Backpack computer. (Hewlett-Packard, nd)

is designed to run VR applications specifically. A VR Ready computer is a machine of which its components are powerful enough to run XR applications and which has enough ports to connect all the components, like the HMD. The combination of its technical specifications and being a wireless PC, makes it the perfect computer for this project.

4.3 Deployment diagram

Looking at the UML deployment diagram (Figure 4.20), it is clear that there are eight types of devices (the firehose nozzle consists of two separate devices) that have to be deployed to make the system work properly.

The main device is the VR ready PC that runs Windows 10 and has the XRFirefighting application installed on it. Then there is the HTC VIVE pro Linkbox that is connected to the PC using two types of connections, an USB 3.0 cable and a DisplayPort cable that is plugged into the GPU. There are four types of devices connected to the VIVE Linkbox, there is the HTC VIVE pro VR headset that is connected using its own cable, there are two SteamVR base stations 2.0 that are connected wirelessly, one HTC VIVE pro Wand that is connected wirelessly, and one HTC VIVE Tracker 2.0 that is also connected wirelessly. The VIVE tracker is physically connected by magnets to the firehose nozzle which also contains the Arduino and all the sensors. The Arduino sends the processed sensor data using Wi-Fi via the User Datagram Protocol (UDP) to the PC. Lastly, there is the ZED Mini stereo camera that is mounted on the VR headset and is connected directly to the PC using a USB 3.0 connection.

4.4 Hardware components and schematics

The firehose nozzle is a wireless device. This allows the user to move around freely without tripping over electrical cords. This also enhances the XR experience, since the user does not have to consider any real-world obstacles when moving around. To achieve this, a few extra constraints

had to be taken into account while designing this measuring system.

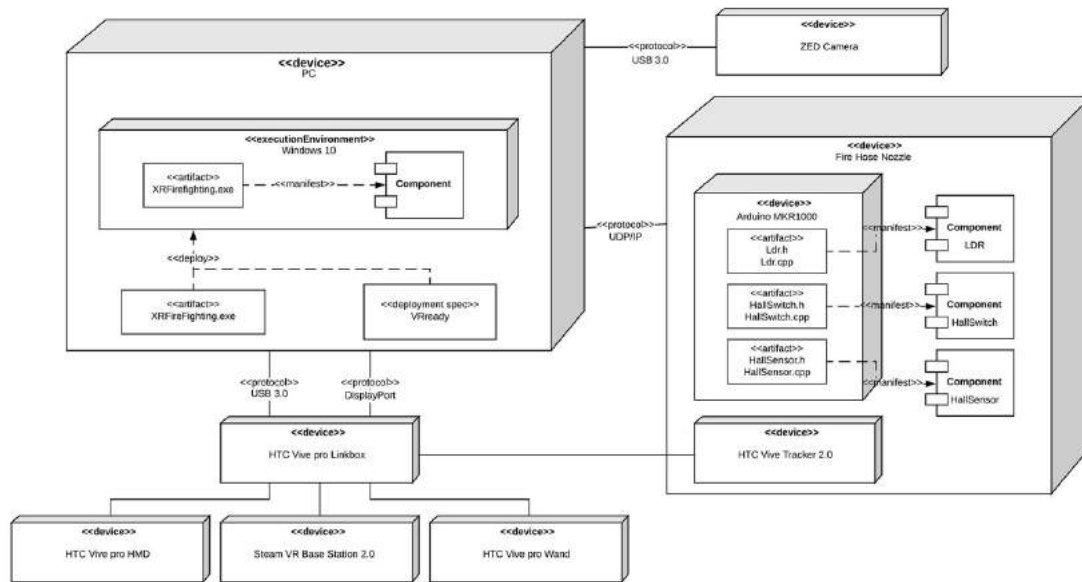


Figure 4.20: Deployment diagram of the application.

4.4.1 Arduino

The board used for this project is an Arduino MKR 1000 (Figure 4.21). The board has a built-in LiPo connector and charging circuit, and a built-in Wi-Fi module. It can also be powered by an external supply of 5V, which will charge the battery when both are connected. The external supply can be connected to the micro USB port or the Vin pin. The battery has to be a single cell 3.7V LiPo battery with at least 700mWh. The board itself operates at 3,3V. The Wi-Fi module of the board supports the 802.11 b, g and n standards. (Arduino, nd)



Figure 4.21: Arduino MKR1000. (Arduino, nd)

A LiPo is a lithium-ion polymer battery. These rechargeable batteries can store a lot of power

but are lightweight and can be fairly small. This makes this kind of battery ideal for this project compared to portable power banks which are a lot bigger. The fact that the battery is so small allows it to sit inside of the measuring head. This prevents the user from touching and breaking it during the game. The battery used for this project is a single cell, 3,7V battery with a rating of 700mwh.

UDP was chosen to send the data to the application. UDP is a connectionless communication protocol which means that there is no established connection between communicating devices as a consequence, there is less overhead. A connection-oriented protocol, like the Transmission Control Protocol (TCP), uses a three-way handshake, but this creates a lot of overhead. With UDP, packets are just sent to a certain IP address even when that address is not available on the network. The reduction of overhead makes the exchange of data faster, which results in minimal response time of the virtual model of the nozzle.

4.4.2 Sensors

The firehose nozzle is equipped with six sensors: two Hall sensors, three Hall switches and an LDR. All these sensors are connected to an Arduino which sends the received data to the application. Each sensor, its technical specifications and circuit is explained more in depth in the following section.

4.4.2.1 Hall sensor

There are two analogue Hall sensors that measure the state of the flow rate selector. When the flow rate increases, the distance between the measuring head and the actual nozzle becomes bigger. We attached two magnets to the nozzle itself. These magnets are positioned right underneath the sensors. The sensors sense a change in magnetic field when the magnets move.

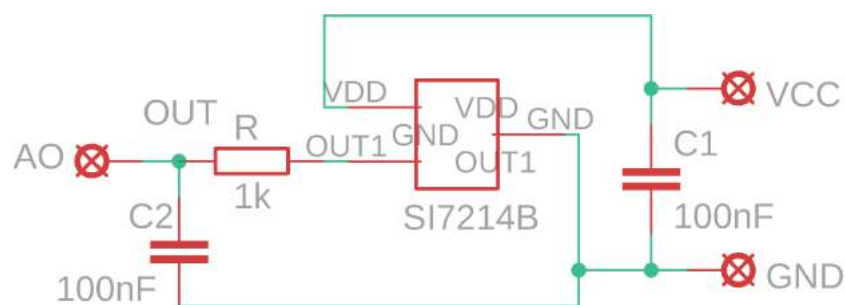


Figure 4.22: Electrical circuit of the Hall sensor.

The analogue Hall sensor used for this project is the surface-mount device (SMD) version of the Si7211 from Silicon Labs. The recommended power supply is 2,5 V to 5,5 V. The output is half the power supply when no magnetic field is detected. When a large negative magnetic field is sensed, the output voltage drops to zero. When a large positive magnetic field is sensed, the output voltage

is close to the voltage of the power supply. (Silicon Laboratories Inc., 2019)

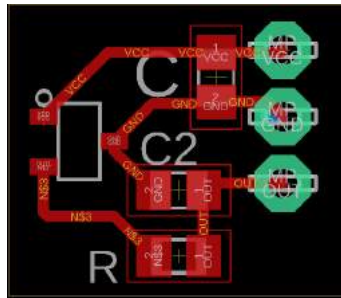


Figure 4.23: Hall sensor PCB design.

The circuit of this sensor is shown in Figure 4.22 and its PCB design is shown in Figure 4.23. A decoupling capacitor is connected to the power supply to prevent spikes from damaging the sensor. To prevent unwanted noise from entering the Arduino, a low-pass filter is put at the output of the circuit.

4.4.2.2 Hall switch

To detect the selected beam type, there are three digital hall switches and one magnet. When the user rotates the beam type selector, the magnet will move underneath a certain switch. This switch detects the magnet and sends this data to the Arduino.

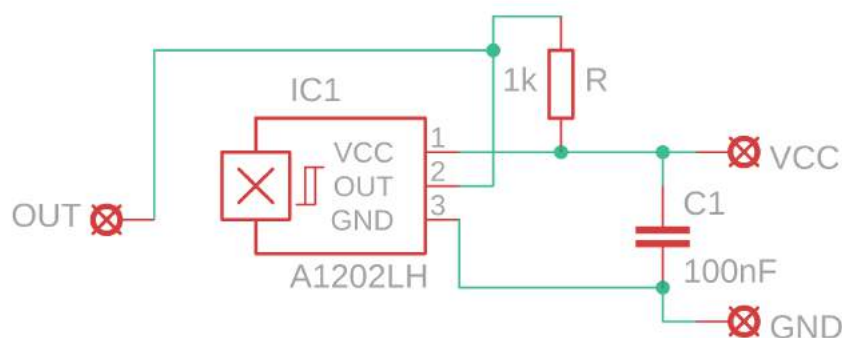


Figure 4.24: Electrical circuit of the Hall switch.

The hall switch that is integrated in this system is the SMD version of the DRV5033 from Texas Instruments. It is a digital omnipolar-switch Hall sensor. An omnipolar switch was chosen so it could detect a change in the magnetic field for both the positive and negative pole. This ensures that if something would go wrong and the magnet would be reversed, it would not influence the system because a change would still be detected.

The sensor can be powered by a power supply of 2,5 V to 38V. The sensor requires an external pull up resistor in order to be read properly. When a strong magnetic field is sensed, the output will be

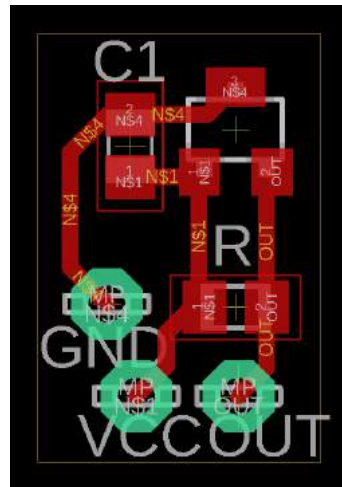


Figure 4.25: Hall switch PCB design.

pulled low. Hysteresis is implemented so noise does not have an effect on the output. The circuit and PCB are shown in Figure 4.24 and Figure 4.25 respectively. To prevent spikes in the voltage supply from damaging the circuit, a decoupling capacitor is used. (Texas Instruments, 2016)

4.4.2.3 Light dependent resistor (LDR)

To measure the position of the big black lever, a light dependent resistor is used. This sensor sits at the inside of the nozzle. On the insert of the backplate there are three LEDs. Between the LDR and the LEDs, there is an internal valve. When the lever moves from the closed position to the open position, this valve opens (Figure 4.1, nr. 2). This allows the light from the LEDs to pass through and reach the LDR. The more the lever is opened, the more light passes through and reaches the LDR. This will make the resistance of the LDR drop. The more light that reaches the LDR, the lower its resistance will be.

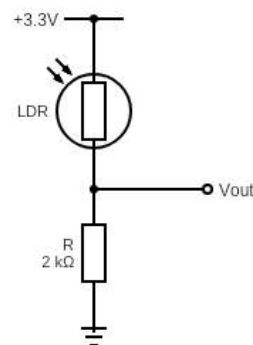


Figure 4.26: Electrical circuit of the LDR.

The LDR used for this project is the TO-18 Hermetic Photocell NSL-06S53 from Advanced Pho-

tonix, Inc. The diameter is around 5,5 mm which makes it small enough to fit inside the hose. At 100 lux it has a light resistance (R_L) of $5k\Omega$. When all light is removed, it has a resistance (R_D) of $20M\Omega$. (Advanced Photonix Inc., 2014) After doing some measurements, the light resistance seemed to be around $1,13k\Omega$ instead of $5k\Omega$.

Figure 4.26 shows the circuit of the LDR. It is connected in series with a $200k\Omega$ to the supply voltage of the Arduino, which is $3.3V$. The choice for a $200k\Omega$ resistor in series is further motivated below in the calculations for the output voltage (V_{OUT}) in a light environment and a dark environment.

$$R_L = 1,13 k\Omega, R_D = 20 M\Omega \text{ and } V_{CC} = 3,3V$$

Using Ohm's law:

$$I_L = \frac{V_{CC}}{R_L + R} = \frac{3,3V}{1,13k\Omega + 200k\Omega} = 16\mu A$$

$$\Rightarrow V_{OUT} = R * I_L = 200k\Omega * 16\mu A = 3,28V$$

$$I_D = \frac{V_{CC}}{R_D + R} = \frac{3,3V}{20M\Omega + 200k\Omega} = 163nA$$

$$\Rightarrow V_{OUT} = R * I_D = 200k\Omega * 163nA = 3,3mV$$

As can be seen in the calculations, the output voltage is almost equal to the input voltage when the LDR is lit. The ADC of the Arduino will output 1024. When the LDR is not lit, the output voltage is almost zero. The ADC of the Arduino will output zero. With a $200k\Omega$ resistor, the full output range (0 - 1024) of the ADC is used while still maintaining a low power consumption.

4.5 Software design

4.5.1 VR simulator

By using the VR simulator, a trainee can be immersed in a complete virtual environment. This environment is a specifically designed scenario. Later on, more scenarios can be added to be able to train in different environments. A drawback of this system is that each scenario has to be designed specifically to the needs of the Navy. Therefore, in the future, they will have to rely on a third party to design prefabs and scenes that can be added to compose a specific scenario.

When looking at the use case diagram for the VR simulator (Figure 4.27) two primary actors can be distinguished. One is the instructor, the other is the trainee. The primary actors each have a range of possible interactions they can perform with the system. The instructor can place a fire on a flammable object, he can grow a fire if he clicks on a particular fire that has not reached its maximum size and he can start the game. After starting the game, he has to give the VR headset to the trainee. If the instructor wants to setup a training session for himself/herself, he becomes the trainee. The trainee has his own range of actions the first one being, extinguishing the fire. This will diminish the fire's health and kill the fire if its health has reached its minimum health threshold. The trainee can also perform several actions on the firehose nozzle which include, changing the beam type which will change the shape of the water beam, change the flow rate which will change the amount of particles emitted, open the firehose nozzle which will start the emission of water

and finally the trainee can also close the firehose nozzle which will stop the emission of water. All interactions on the firehose nozzle are automatically logged into a text file that can be viewed afterwards.

Fires can be spawned by the instructor using the VIVE Wand and the VIVE VR headset or by using a computer screen and a mouse. When using the Wand, a laser pointer appears in VR and the instructor can use the laser pointer to indicate the position in which he wants to spawn a fire. This position must be on a flammable object otherwise no fire will be spawned. Spawning fires using the computer screen is as simple as clicking on a flammable object. Currently, only class-A fires are allowed which means, only solid objects are marked as flammable. Keep in mind that a class A-fire does not include metals.

Fires that are spawned grow automatically with time until they reach their maximum size. The maximum size depends on the object the fire is spawned upon. Fires can also instantiate new fires which can grow and instantiate other fires. This process keeps repeating itself to spread the fire over all flammable objects in the neighbourhood.

In the real world, heat between non touching objects is mostly transferred by radiation. But calculating the energy that is transferred through thermal radiation is a complex process. For that reason, it was chosen to simulate the heat transfer between non touching object by using conduction between the object and the hot air around the fire and multiplying this with a constant. This is not physically correct but results in a nice visual effect that looks realistic if the constant is chosen correctly. Studying the exact behaviour and propagation of fire is part of an advanced Master's programme in fire safety engineering and falls out of the scope of this thesis. Heat transfer through convection is also not implemented because its implementation is too complex and resource intensive.

4.5.2 MR simulator

The main reason why an MR application is preferred over a VR application is because this allows the Navy to train in very specific environments without any rooms that need to be modelled. The Belgian Navy will benefit most from a training tool that trains and prepares the crew for real-life scenarios. With an MR application, the system can be setup in any compartment of the ship and training can begin. This is a huge advantage since not every compartment of the ship needs to be designed by a developer. Another major advantage is that it is possible to use props, like a door, that they already have. This will make the training of opening a door, for example a lot more effective. The way one opens a door of a room that is on fire should be done following a specific procedure. The door cannot just be opened. Opening a real door instead of pressing a button on a controller (as would be done in VR) enhances the immersion of the system and effectiveness of the training.

Figure 4.28 shows the use case diagram of the MR simulator and all possible interactions between the users and the system. There are two actors, the instructor and the trainee. Each MR training starts with the instructor placing fires. The instructor himself decides how many fires he places. He can also make existing fires bigger by clicking on them. When the instructor is finished placing fires, he can start the game. After receiving the headset, the trainee can now start his training. The

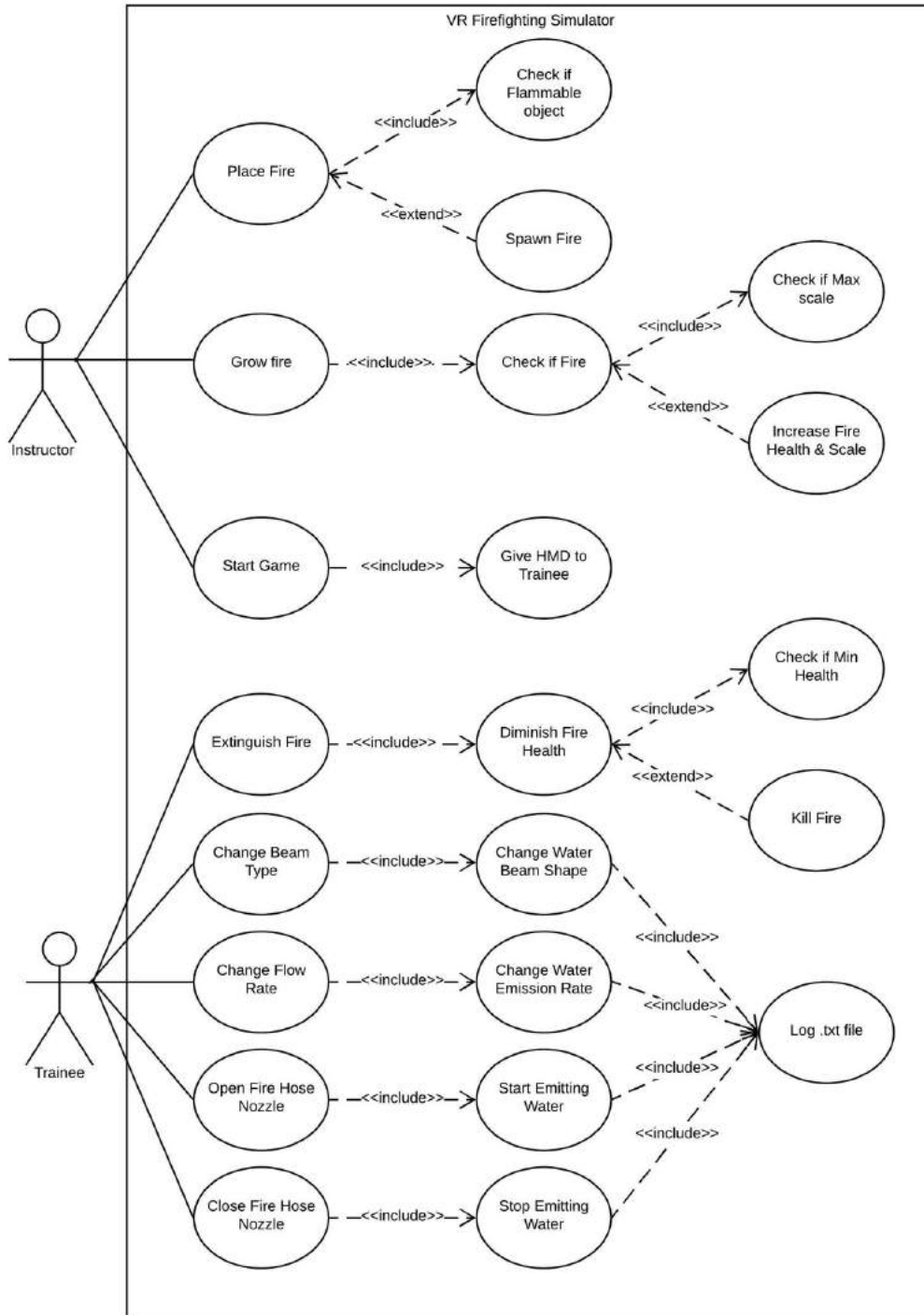


Figure 4.27: Use case diagram of the VR simulator.

trainee can extinguish the fire using one of the available fire extinguishers. Each physical action performed on the extinguisher or nozzle is automatically logged into a text file.

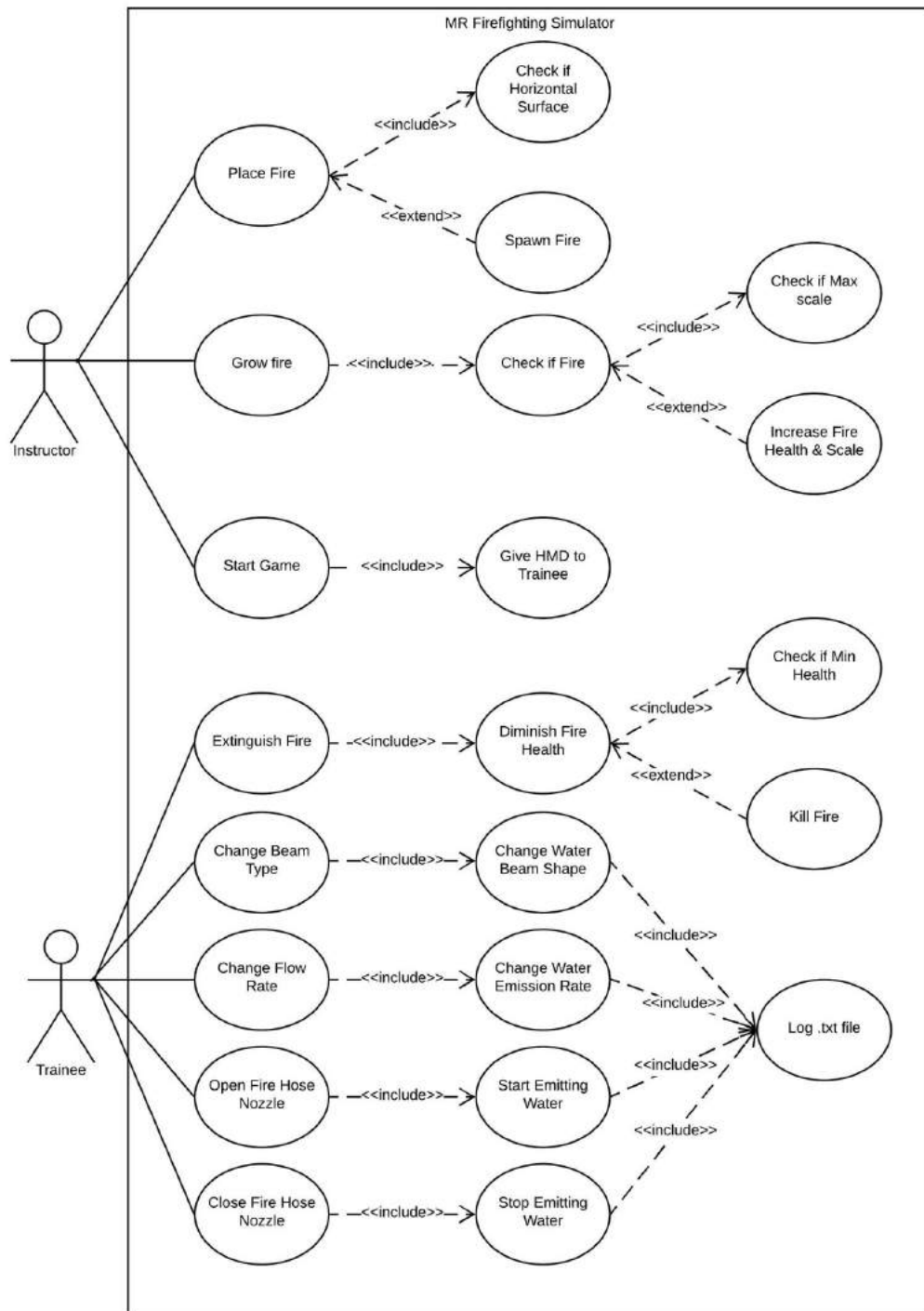


Figure 4.28: Use case diagram of the MR simulator.

The instructor should have the possibility to indicate where a new fire needs to be spawned or which fire needs to increase. To accomplish this task, the default VIVE controller is used with an

added virtual laser pointer. The instructor points the laser at a certain position on a horizontal, pink highlighted surface. When he pulls the trigger of the controller, he spawns a new fire at that position. When he points the laser on an existing fire and pulls the trigger, that fire increases in size. After placing all the fires, the instructor can continue to the next phase of the game, which is extinguishing the fires, by pressing the menu button on the controller. This is the button above the trackpad. The pink highlighted planes will disappear, and the extinguishers/ nozzle will start to work.

To extinguish the fire, the trainee uses one of the available extinguishers. For this thesis, only the firehose nozzle is implemented. When one opens the nozzle, a virtual water beam appears at the front of the firehose nozzle. It will seem like there is water coming out of it. When the water beam is pointed at the fire, it will start to decrease in size until it goes out.

Chapter 5

Development and Implementation

This chapter discusses the implementation of each part of the system. It starts with an overview of the used implementation environments. Followed by a section about the integration of the hardware components. Next, there is a section on how the VR and MR simulators are programmed in Unity. At the end of this chapter an explanation is given about how object detection should be implemented in the system.

5.1 Implementation environment

Different implementation environments were used to create the system. This section gives an overview of which implementation environments were used. This ranges from, the 3D modelling software used all the way to which packages must be installed to implement object detection.

5.1.1 3D modelling and printing software

Autodesk Fusion360 (version 2019.2.0.7824) is the main 3D modelling software used to create the printed parts. After designing the parts, they were sliced using the Ultimaker Cura software (version 4.5.0). A modified Creality CR20 3D printer was used to print all the parts. The modifications to the printer were done to ensure quality 3D prints from a budget 3D printer. The modifications include an upgraded magnetic print bed that improves bed adhesion, an upgraded extruder module that ensures consistent filament feeding to the hot end and thus prevents over-extrusion or under-extrusion and finally an upgraded hot end with nozzle that limits the risk of clogging and increases precision.

5.1.2 Unity

The game engine used for this project is Unity (version 2019.3.10f1) by Unity Technologies. A game engine is software that facilitates game developers to build games. It contains all the necessary tools to easily create a game. Unity is one of the most popular game engines, especially

for XR game development. Unity works with scenes to which game objects are added. Custom behaviours can be assigned to these game objects by writing C# scripts. (Unity Technologies, nd) For Unity to be able to work with the HTC VIVE Pro headset, the SteamVR plugin needs to be downloaded and imported into the project. The new SteamVR Unity plugin, SteamVR 2.0, comes with a whole new input system. Which allows developers to define custom action sets for the controller. An action set is a collection of actions that can be mapped to the buttons of the controllers, using the bindings UI.

5.1.3 ZED Camera

The camera that was used for this thesis, is the ZED Mini from Stereolabs (Figure 5.1). This camera is developed for AR applications specifically and can be attached to a VR headset. The depth range of this camera is 0,1 to 15m and it is equipped with two motion sensors, a gyroscope and an accelerometer. The field of view is 90° (horizontal) x 60° (vertical), which is smaller than the human field of view. When the image is displayed in an HMD, black bars are added to each side of the image. As a result, the player will have a limited view. The image sensors have a 16:9 format and at a resolution of 1344x376, the ZED Mini is able to deliver up to a 60 fps for XR. The camera is connected by a USB 3.0 type C cable. (Stereolabs Inc., ndc)

The camera comes with multiple third-party integrations, like a Unity plugin. To use this plugin, the ZED SDK needs to be installed too. The main components of the Unity plugin are the two camera prefabs. The first prefab, the *ZED_Rig_Mono*, is used for AR applications that do not require an HMD. The second prefab, the *ZED_Rig_Stereo*, is used in MR applications. This prefab can only be used with an HMD and replaces the SteamVR Camera Rig. It outputs two images, one for each eye.



Figure 5.1: ZED Mini (Stereolabs Inc., ndc)

The plugin also contains many useful scripts. The most important ones being the *ZEDPlaneDetectionManager* and the *ZEDControllerTracker_DemoInputs*. The *ZEDPlaneDetectionManager* detects flat surfaces. The detected surfaces can be highlighted in pink. The plane detection manager can also detect the floor plane. This plane will be highlighted in a light blue colour. The user can select which detected planes should be visible in the scene view and/or in the game. The *ZEDControllerTracker_DemoInputs* script handles the movements of the controller. It also handles the user's actions, like button presses and the trigger. To handle the input of the controllers, custom bindings can be made in the SteamVR bindings UI. The ZED plugin for Unity comes with its own action set. These actions can be assigned to each input of the tracker.

The ZED Mini can be attached to the front of the headset. The mount that comes with the camera

is not compatible with the HTC VIVE Pro, so an additional mount needs to be printed. Stereolabs provides an STL file with the 3D model of the mount on their website. This file should be sliced before it can be printed, as explained in section 5.1.1.

5.1.4 Python and TensorFlow

TensorFlow is a package that can be imported in python scripts to easily develop machine learning algorithms. In this thesis, an attempt to implement object detection has been made. To implement object detection, the TensorFlow Object Detection API is used. This API is not yet compatible with TensorFlow 2.0, that is why TensorFlow 1.15.0 was used. Python also needs to be installed in order to use TensorFlow. For this project, Python 3.7 was used in combination with Anaconda.

5.2 Arduino and sensors

This chapter describes how all data from the sensors is processed before it is sent to the main application on the computer. It gives an overview of how the sensors are connected to the micro-controller and the libraries included in the Arduino sketch.

5.2.1 The board

Figure 5.2 shows how each sensor and PCB is connected to the Arduino MRK 1000. The output of the LDR is connected to pin A0. The PCBs of both analogue Hall sensors are connected to pin A1 and A2. The PCBs of the digital hall switches are connected to pins D0, D1 and D2. The Vcc of each PCB is connected to the power supply of the board which is 3,3V. All grounds of the PCBs are connected to the ground of the board. After the analogue pins are read, the 10-bit ADC converts the voltage to a digital value in the range 0 to 1023.

5.2.2 The software

An Arduino program consists of two major parts, the setup and the loop. The setup is called only once, right at the beginning of the execution of the program. After that, the loop is executed continuously. In the setup, the agent type is set. This is done with an integer, zero for a firehose nozzle, one for a powder extinguisher and two for a CO_2 extinguisher. Next, the board will try to connect to the Wi-Fi network with the given SSID and password. It will try to make a connection every second until it succeeds. In the next step, the UDP connection is initialized. In the loop, every sensor is read. To read the values of the sensors, four custom libraries were written. Three libraries are responsible for reading each kind of sensor, and one to process the data from the sensors. The LDR library is responsible for reading the LDR sensor output. After the output of the LDR is read, it is pushed to the end of an array and a running average is calculated and returned. The user can define the length of the array, and so the number of samples to calculate the running

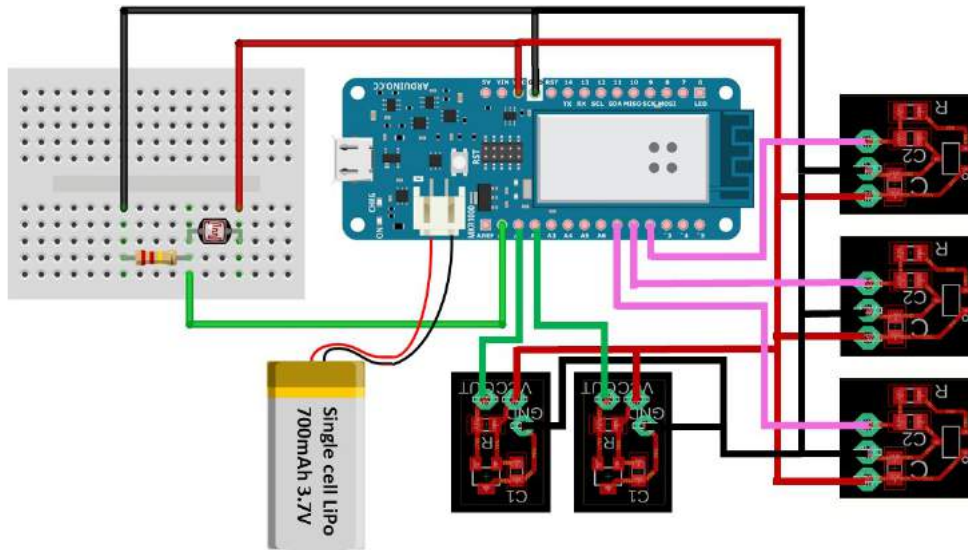


Figure 5.2: Connection scheme of the Arduino

average over, in the beginning of the Arduino code. The need for a running average comes from the observation that the output voltage of the sensor is not very accurate. Its digital output value read by the Arduino can fluctuate with ± 5 units.

The libraries responsible for reading the analogue Hall sensors and hall switches both contain a function to read the output of the sensor.

The fourth library is the Nozzle library. This library is responsible for processing the received data and preparing it to be sent to the application. The LDR value is converted to an angle between 0 and 90 degrees. This is the range of the actual lever. For the conversion, a function was found through an experimental method. For different positions of the lever, the output value was read multiple times and the average was calculated. From the calculation, we could derive the function in equation 5.1.

$$f(x) = \begin{cases} 0 & x < 17 \\ 0,0441 * x + 29,051 & 17 \leq x < 1010 \\ 90 & 1010 \geq x \end{cases} \quad (5.1)$$

The average of the measurement of two analogue Hall sensors is used to increase the accuracy of the flow rate selector's position reading. The average of both sensors' measured values is larger than 300, so this is subtracted from the average. The reason for doing this is that now, no average value is larger than 255 and the value can be sent within one byte. Further processing of this value is done by the XR application. For all eight settings of the flow rate selector, a range of values is determined. The calculations of these ranges are shown in appendix A.

The beam type is detected using three digital hall switches. The output of these sensors is high if there is no magnet underneath it and low if there is a magnet present. Depending on the selected

beam type the output will be an integer between zero and two.

Besides the use of custom written libraries, certain standard libraries were used too. The *WiFi101* library is needed to connect the Arduino to a Wi-Fi network. The *WiFiUDP* library allows to set up an UDP connection between the Arduino and the application.

All the components of the virtual nozzle follow the real nozzle's manipulations in real time. This means that the server has to process the received data and convert it to movements on the screen. Processing the data in the Arduino program reduces the load on the server of the application.

After reading all sensors and processing the data, the UDP packet is prepared for transmission. To reduce the load on the server, a packet will only be transmitted when it is different from the previous sent packet. The packet is four bytes long. The agent type fills the first byte. For now, there are only three options so, only the two least significant bits are used. But this can be extended in the future by using the remaining six bits as flags. The lever position fills the second byte. The flow rate and beam type fill the third and fourth byte respectively. Figure 5.3 shows the structure of the packet.

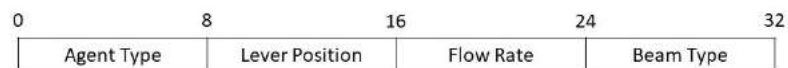


Figure 5.3: Structure of the payload of the UDP datagram

5.3 Implementation in Unity

The main application is made in the Unity game engine using visual scripting and coding in the C# programming language. C# was used to program the logic of the simulator, like how fires spread and how they interact with water, but it was also used to program the networking part of the project. Most prefabs used to create game objects can be found for free on the Unity asset store. Except the firehose nozzle (Figure 4.18) which was created by One Bonsai.

5.3.1 General code

The XR simulator application is divided in two parts, the first one being the VR simulator and the second one being the MR simulator. Both simulators have similarities and thus share some of the same classes and methods.

5.3.1.1 UDPServer

One of the shared classes is *UDPServer*. This class is responsible to collect and process the data received by its clients and relay the information to the correct game object in the scene to update the object's state.

A separate thread is created to collect the incoming data via the method *ReceiveData*. This thread will handle all the incoming 32-bit packages. All packages bigger or smaller than 32 bits are discarded because this would imply a faulty package. Once the packet is received and the size is checked, it is sent to the main thread. The main thread will use the *CreateAgent* method (Listing 5.1) to send the message to the correct game object in the scene or it will instantiate a new game object if the client is connecting for the first time. The type of game object that has to be instantiated depends on the information contained in the package. Each UDP package contains a tag that is the client's agent type (Figure 5.3) which is used to select the right extinguishing prefab (Listing 5.1, line 10). In the current version of the simulator, there are three possible game objects, there is the firehose nozzle, the CO_2 extinguisher and the powder extinguisher. These three types of extinguishing agents can be spawned, but only the firehose nozzle will work in the current state of the simulator. This is because there is no tangible CO_2 or powder extinguisher with sensors connected to it. Thus, no sensor data is being sent to the server. The option to have multiple extinguishers is added to this version of the simulator to show its extendibility which is the third non-functional requirement of the Navy.

Once the game object has been instantiated, an available VIVE tracker has to be linked with the object and its state has to be updated with the received information. The VIVE tracker is used to track the movement of the object in three-dimensional space and mirror these in the virtual scene. To attach the tracker to an object in the scene, an empty parent object is created. This parent object will have the tracker linked to itself and will encapsulate the object that has to be tracked. It is important to know that no game object will be instantiated if there is no tracker available (Listing 5.1, line 28). Thus, it is crucial to check whether at least one tracker is on and connected to SteamVR. To ensure the tracking of the virtual object happens accurately, the Tracker object must be placed correctly relative to the game object in three-dimensional space. To achieve the right placement and correct tracking, the tracker object is kept in the local origin and an offset is applied to the game object's position and rotation (Listing 5.1, lines 48 and 49). The offsets needed can be found in the *AgentInteraction* instance of the game object, by using the methods *mGetTrackerRotOffset* and *GetTrackerPosOffset*.

Updating the state of the game object happens in the *CreateAgent* method by calling the *PacketTranslator* method of the *AgentInteraction* instance that is attached to the game object (Listing 5.1, lines 62 and 70). If the object does not have to be instantiated because it already exists, its state is simply updated in the same way as for a new object.

The reason why the information found in the secondary thread is sent to the main thread is because Unity does not allow a secondary thread to instantiate objects. A *Dispatcher* design pattern (Dupire and Fernández, 2001) is used to send the information from the secondary thread to the main one. The *Dispatcher* class implements a singleton pattern to ensure that there is only one *Dispatcher* object for the whole server. This *Dispatcher* object will save a list of all the actions from the secondary thread that have to be executed by the main thread. These pending actions will be invoked by the main thread in the *Update* loop of the *UDPServer* object.

Working with the dispatcher also allows the server to handle multiple clients at the same time without dropping any of the received UDP packages. Being able to handle multiple clients at once

means that the system is ready for a user with multiple extinguishers. It also allows the Navy to upgrade the system to a multiplayer version, where every player has his own extinguishing agents. This is also in line with the Navy's third non-functional requirement.

```
1 // Creates a extinguisher gameobject if it does not exist and updates the object
  with the new data
2 private void CreateAgent(IPEndPoint clientEndpoint, byte[] data)
3 {
4     try
5     {
6         //If client connects for the first time a new GameObject is made and the
          sending IPadress is linked together in the dictionary
7         if (!clients.TryGetValue(clientEndpoint.Address, out GameObject
          extinguisher))
8         {
9             GameObject prefab;
10            Agent agentType = AgentInteraction.GetAgentTypeOutOfData(data);
11            switch (agentType)
12            {
13                case Agent.HOSE:
14                    prefab = prefabHOSE;
15                    break;
16                case Agent.POWDER:
17                    prefab = prefabPOWDER;
18                    break;
19                case Agent.CO2:
20                    prefab = prefabCO2;
21                    break;
22                default:
23                    Debug.Log("ERROR: Agent type is not recognized for the following
                      client: " + clientEndpoint.Address);
24                    return;
25            }
26
27            // Check if a tracker is available to attach to the agent, so the it's
          movement can be tracked in 3D space
28            Tracker tracker = GetAvailableTracker();
29            if(tracker == null && useTrackers)
30            {
31                Debug.Log("No tracker available"); // If no tracker is available
          (because they're all used or none are connected) the object
          isn't added to the scene
32                return;
33            }
34
35            tracker.Available = false; // Set tracker is in use
36            // Create gameobject
```

```
37     extinguisher = Instantiate(prefab, new Vector3(0f, 0f, 0f),
38         Quaternion.identity);
39
40     // Create empty game object that is linked with the tracker that will be
41     // used as a parent
42     GameObject trackedParent = new GameObject();
43     trackedParent.AddComponent<SteamVR_TrackedObject>();
44     // Link tracker to parent
45     SetTrackedParent(trackedParent, tracker);
46     //Set name seen in the hierarchy of the parent game object
47     trackedParent.name = agentType.ToString();
48     // Set parent (empty object with tracker, it moves with the tracker, so
49     // all the objects it contains will move with it)
50     extinguisher.transform.parent = trackedParent.transform; //
51     // trackedParent is a gameobject while one expects a transform that's
52     // why .transform is added
53     extinguisher.transform.localEulerAngles =
54         extinguisher.GetComponent<AgentInteraction>().GetTrackerRotOffset();
55     extinguisher.transform.localPosition =
56         extinguisher.GetComponent<AgentInteraction>().GetTrackerPosOffset();
57
58     // Add IPAddress as key and new gameobject as value to the dictionary
59     // clients
60     clients.Add(clientEndpoint.Address, extinguisher);
61     // Set the agent type field of the created extinguisher
62     extinguisher.GetComponent<AgentInteraction>().SetTracker(tracker);
63     // Set the agent type field of the created extinguisher
64     extinguisher.GetComponent<AgentInteraction>().SetAgentType(agentType);
65     // Set IP address in weapon component
66     extinguisher.GetComponent<AgentInteraction>().SetIPAddress(clientEndpoint.Address);
67     // Make a WriteTextIO instance to log the data received by the weapon in
68     // a text file
69     extinguisher.GetComponent<AgentInteraction>().SetTextIO(agentType,
70         clientEndpoint.Address);
71     // Modify fields of the new object by the data received by the client
72     extinguisher.GetComponent<AgentInteraction>().PacketTranslator(data);
73     // Add timestamp of last weapon update to the dictionary timestamps
74     DateTime lastUpdate =
75         extinguisher.GetComponent<AgentInteraction>().GetLastUpdate();
76     timestamps.Add(clientEndpoint.Address, lastUpdate);
77 }
78 else
79 {
80     // Update extinguisher with new data
81     extinguisher.GetComponent<AgentInteraction>().PacketTranslator(data);
82     // Update timestamp of last extinguisher updates
83     DateTime lastUpdate =
```

```
    extinguisher.GetComponent<AgentInteraction>().GetLastUpdate();
73     timestamps[clientEndpoint.Address] = lastUpdate;
74 }
75 }
76 catch (Exception err)
77 {
78     print(err.ToString());
79 }
80 }
```

Listing 5.1: *CreateAgent* method of the *UDPServer* class.

5.3.1.2 AgentInteraction

The *AgentInteraction* class is an abstract class that serves as a base class for different types of extinguishing agent classes. Every extinguishing agent type needs its own derived class to be able to update and keep track of the game object it is linked to. This is because every type of extinguishing agent has different controls and movable pieces. For example, a firehose nozzle has three controls with which it can be manipulated. It has the beam type selector, the flow rate selector and the lever. But a CO_2 or Powder extinguisher only have one control, the lever. The base class only contains the methods that are used in every extinguishing agent class. All the agent specific methods are implemented in the derived classes.

An example of a method that is needed for every type of extinguisher is the *SetLever* method. This is an abstract method that is implemented in the derived classes. It is responsible to update the position of the lever and turn the emission of the agent on and off. The lever value is the only argument for the method, and it depicts the real position of the lever on the tangible object. The way that the real-life position or rotation is translated to a position or rotation for the virtual lever, is implemented in this method.

The *PacketTranslator* method is a fully implemented base class method. This method is called by the *UDPServer's CreateAgent* method every time a UDP packet has to be processed. The *PacketTranslator* will update the state of the extinguishing agent it is called upon. This method also contains the call to the *WriteTextIO* object to log the updated state of the extinguishing agent in a text file.

Every extinguishing agent type will need its own *AgentInteraction* variant that inherits from *AgentInteraction*. Each variant will have to implement the methods that are needed to update the state of their specific extinguishing agent based on the sensor data received.

The *AgentInteractionHOSE* class is a child class of *AgentInteraction* that implements all the methods needed to update the state of the lever, beam type selector and flow rate selector based on the received UDP packet. These methods also influence the water particle system that emits particles out of the firehose nozzle. Changing the beam type for example will change the particle systems shape. While changing its flow rate will alter the particle systems emission. Changing the position of the lever will influence the throughput of water particles out of the firehose. This class also

contains the specific offsets for connecting the VIVE tracker to its instantiated prefab.

5.3.1.3 WriteTextIO

The *WriteTextIO* class is used to log the data of all the clients of the server. Every client has its own *WriteTextIO* object that creates a separate text file to log the data with a timestamp. This feature can be used by instructors to review or replay the manipulations executed on the extinguisher during the training session. Although additional software would simplify the reviewing process.

5.3.1.4 Tracker

The *Tracker* class is used to check if a tracker is connected. In the server there are four slots to fill in the serial number of different trackers. This gives the user the freedom to choose which tracker to use. If one tracker has to be recharged, the user can swap trackers and start a new training session. Keep in mind that the four slots do not have to be filled. Only one tracker is necessary to use the simulator, so at least one slot should be filled.

When the *UDPServer* is instantiated, it goes through the array with the four slots and creates *Tracker* object for the non-empty slots. Each *Tracker* object represents a physical tracker and contains the serial number of that tracker, the tracker's ON/OFF-state and if the tracker is still available. Being available and ON does not mean the same thing in this context. If a tracker is ON, it is connected to the application. If a tracker is available that means that the tracker is connected to the application and it is not linked with any other game object in the scene. When a tracker is linked with a game object, it is indicated in the tracker field of the *AgentInteraction* instance attached to the game object.

5.3.1.5 Water

The *Water* game object, which is attached to the water emitter of the firehose nozzle, is a modified version of the *Shower* prefab from Unity's Particle Pack. The original *Water* game object consists of a parent game object to which a particle system, the beam itself, is attached. This parent game object has two child game objects, *Splash* and *Ripple*, which also have a particle system component. The original prefab is modified, so it would better resemble a firehose.

First, the rotation of the particle is changed so it sprays forward instead of downwards. Second, the number of particles that are emitted is increased to make it look more realistic. In game, the parameter that relates to the number of emitted particles can be changed by turning the flow rate selector. A higher flow rate results in a higher emission. The third modification that is made is the shape of the particle system. The shape of the particle system is a cone, whose angle changes when the beam type selector changes.

Last, the *Water* or *MRWater* script is added to the parent game object. This script describes the behaviour of the water. The method *OnParticleTrigger* is called every time a trigger event with a water particle happens. This method is used to decrease the fires' health when they are hit by

water particles. In order for the particle system to generate trigger events, the trigger module of the particle system should be enabled. The trigger module needs a list of trigger colliders. Only when a particle enters one of those colliders, a trigger message is generated. The *OnParticleTrigger* method makes a list of all the particles that entered one of those colliders. For each particle in the list, the program verifies which fire it entered. If it hit a certain fire, the fire's health decreases with a constant value. The code is shown in Listing 5.2. The list of these colliders is updated every frame.

```
1 private void OnParticleTrigger()
2 {
3     //Get all entered particles that entered one of the colliders in the list of
4     //the trigger module of the particle system
5     List<ParticleSystem.Particle> enteredParticles = new
6     List<ParticleSystem.Particle>();
7     int enterCount =
8     waterPS.GetTriggerParticles(ParticleSystemTriggerEventType.Enter,
9     enteredParticles);
10
11     //Get all fires
12     GameObject[] fires = GameObject.FindGameObjectsWithTag("Fire");
13     //for each particle in the list, check which fire it hit
14     foreach (ParticleSystem.Particle particle in enteredParticles)
15     {
16         for (int i = 0; i < fires.Length; i++)
17         {
18             //get the collider of the fire
19             Collider collider = fires[i].GetComponent<Collider>();
20             //check if the particle is inside the collider
21             if (collider.bounds.Contains(particle.position))
22             {
23                 fires[i].GetComponent<Fire>().Damage();
24             }
25         }
26     }
27 }
```

Listing 5.2: *OnParticleTrigger* method of the *Water/MRWater* class.

5.3.1.6 Fire

When a new fire is spawned, a new instance of the fire prefab is instantiated. This prefab is a modified version of the *WildFire* prefab from Unity's Particle Pack. The modifications optimize the prefab for this application and missing features are added, like for example smoke. The original prefab consisted of one parent game object with two child prefabs. Each of these three game objects had a particle system. The first modification is the addition of another child game object which contains the smoke particle system. This smoke particle system is also a modified version of

the *SmokeEffect* from Unity's Particle Pack. The smoke particle system was modified so the smoke goes upwards and does not stay on the ground. The second modification is the restructuring of the prefab. An empty game object is created to which all game objects that contain a particle system are added. The parent game object of the original fire becomes a child object of the new parent too. The new parent game object now has four child game objects. Rearranging the hierarchy of the prefab is needed to scale it properly, so when a fire increases in health, it also grows in size. To simplify the object verification, the tag of the prefab is set to 'Fire'. A box collider is added to the parent game object. This box collider is set as a trigger. The effect of this setting is that other game objects or particles do not bounce off of but go through the collider. When a game object or particle enters the trigger, a trigger event is launched, and both the fire and the intruding game object get a trigger message. The box collider sits at the base of the fire to make the extinguishing process more realistic. In real life, fires should be hit at the base and not at the top of the flames when using water as an extinguishing agent. A disadvantage of using triggers instead of collisions is that a trigger event does not give any information about the objects that caused the trigger event. When a particle collision happens, the method *OnParticleCollision* is called. The parameter of this method is the game object that was hit. Using collisions instead of triggers would make the processing of this event a lot easier but does not give the visual effect that is aimed for because when using collisions, the water particles start bouncing off of the Fire's box collider. At run time, when a new instance of the fire is instantiated, the *Fire* or *MRFire* script is added to the game object to define the behaviour of the fire.

5.3.1.7 FireManager

The *FireManager* class is responsible to keep track of all the Fire game objects in the scene. It stores a reference of all the objects in a list that can be accessed by other classes using the *GetFires* method. The *Water* class attached to the water particle system is an example of a class that uses this method to update its colliders.

Other classes like the *FireSpawner* can modify the list by using the *AddFire* method to add a new *Fire* object to the list. When a *Fire* object is destroyed because it is extinguished, it raises the *KillMe* flag. This Boolean is used by the *FireManager* to know which *Fire* objects have to be removed from its list and have to be destroyed. It is the *FireManager* that is responsible of destroying extinguished *Fire* objects using the *KillFires* method. This method is executed every second as a coroutine rather than being executed in the Update loop. Doing this helps with the performance of the application. Every method in this class will try to access the list of fires, some will even modify it. So, it is important to make sure that the list is not modified by two actors at the same time. Nor that it is read by one actor while the other is writing in it. To ensure thread safety and prevent data corruption, a mutex is used in all the methods. This will lock the list, to ensure that only one actor has access to the list at the same time. An example of how the mutex is used in the *FireManager* can be seen in Listing 5.3.

It is also important to notice that when a *Fire* object is destroyed, its *Smoke* child object is not. The *Smoke* child object is taken out of the *Fire* object before the *Fire* object gets destroyed. This

is done to create the realistic effect of still having smoke lingering around while the fire is already extinguished. The *Smoke* object is destroyed at a later phase by the *DestroySmoke* class that is attached to the *Smoke* object.

```
1 public void AddFire(GameObject newFire)
2 {
3     mutex.WaitOne();
4     fires.Add(newFire);
5     amountOfFires = fires.Count;
6     mutex.ReleaseMutex();
7 }
```

Listing 5.3: Use of a mutex in the *AddFire* method.

5.3.1.8 Menu

The first scene that is loaded when the game is started is the menu screen, called *MainMenu* (Figure 5.4). The menu consists of three buttons: *Virtual Reality*, *Mixed Reality* and *Quit*. When the *Virtual Reality* button is pressed, a new screen, called *VRInfoScreen*, is loaded which contains all the information on how to play the simulation. There are also two buttons, one to go back to the menu and one to start the actual simulation. When *Mixed Reality* is selected, the *MRInfoScreen* is loaded. It looks like the *VRInfoScreen*, but with the information on how to play the MR simulator. When *Quit* is selected, the application closes.



Figure 5.4: Main menu

In Unity's build settings, all scenes are added and receive an index. Using the *SceneManager*, the next scene can be loaded using its index from the build settings.

After the game ends, another menu screen *EndMenu* appears. In this screen, the player can choose to play the same simulator again or to go back to the *MainMenu* screen. Both the VR

and MR simulator have a slightly different version of the *EndMenu*. The end menu contains the after-action report of the training session. For now, only the amount of water used, is displayed.

5.3.2 Implementation of the VR simulator

Implementing a simulator in VR offers a lot of freedom for the programmer. There are almost no limits to what can be simulated other than computational power. This freedom comes from the fact that the virtual environment is not constrained by the real world. The only constraint could be real-world moving space, but even then, there are solutions to circumvent the space constraint by using a teleportation feature. This means that the possibilities to make a firefighting simulator are endless and thus many different approaches can be taken to meet the requirements of the Navy. The implementation proposed in this thesis is just one of the many ways to create a firefighting simulator. A game view of the VR simulator is shown in Figure 5.5.



Figure 5.5: Game view of the VR simulator

5.3.2.1 FireSpawner

Before a training can start, the instructor has to place fires in the scene. This can be done in two different ways. Fires can be placed using the computer itself by clicking on different objects in the scene on which fires should be spawned upon. Or the VR headset and the VIVE Wand can be used to place fires. Both ways of spawning fires are handled by this class.

When using a mouse, the *MouseClick* method is called. When the left mouse button is clicked this method will cast a ray from the position of the camera through the position on the projection plane where the mouse click was registered. By following this ray till it hits an object, a hit point is determined.

Using the hit point, a spawn point is calculated. The spawn point has the same x- and z- coordinates as the hit point but the y-coordinate is equal to the lowest point of the axis-aligned bounding box (AABB) of the object. This ensures that the fire is spawned at the bottom of the object and not

somewhere in the middle. Which gives a more realistic feeling to the phenomena.

When using the VIVE Wand to spawn fires, a laser pointer is displayed in VR. This laser pointer gives a visual queue to the user of where he is aiming at. It shows the user where he is aiming, and he can intuitively select an object on which he wants to spawn a fire. When the trigger of the controller is pulled, the *PointerClick* method is executed. This method will take the point on an object at which the laser pointer was pointing when the trigger was pulled, and it will determine the spawn point in the same way as for the mouse click.

To spawn the actual fires, the *SpawnFire* method is called. This method is responsible to spawn a fire in the spawn point that is passed as an argument. *SpawnFire* will first check what type of object was hit by checking if the object is a Flammable object or a Fire object.

If the object that is hit is a *Flammable* object, then it is checked whether a *Fire* object is already in the neighbourhood of the spawn point. This is done using the *SpawnCheck* method that will check with the *FireManager* if there is a fire in a radius of *MINDISTANCE* from the spawn point. *MINDISTANCE* is a constant that is chosen based on experimental testing. If no fire is in the neighbourhood, a new fire is spawned in the spawn point and its parent is set to the object on which it is spawned. If a fire is already present in the neighbourhood, no fire is spawned to reduce the amount of particle systems in the simulation. This is done because particle systems take up a lot of system resources, so minimizing the amount of particle systems in a scene is beneficial for the performance of the application. Particle systems that are close to each other are not really distinguishable and thus having multiple particle system close to each other does not improve the user experience. Not spawning fires that would be close to other fires will therefore not be a drawback, but an advantage because it limits the amount of particle systems in the scene.

If a *Fire* object is hit, the *Fire* object will become bigger by using the *ClickUpdate* method of the *Fire* script that is attached to the *Fire* object. This is how an instructor can place bigger fires. First, the instructor has to place a default fire before optionally enlarging it by clicking on it. The more he clicks on it, the more the fire will grow until it reaches its maximum scale.

5.3.2.2 FireCore

A *FireCore* is an object that is instantiated by the *FireSpreader* class and it is used to spread fires. This object is a normal C# object and not a "Unity object" that inherits from *MonoBehaviour*. This means that the *FireCore* script cannot be attached to 3D object in the scene. A *FireCore* is an object that keeps track of the temperature in a certain point on a Flammable object. To accomplish this, the *FireCore* uses the physical properties of the flammable object like its thermal conductivity, its temperature, its weight and its heat capacity.

A VR application is already computational expensive thus resources must be spent carefully. When making a firefighting simulator it is better to spend resources on the smoothness of the experience (e.g. achieve higher fps) than on physically correct fire simulations. If the end goal of this project was to build a fire simulator for research purposes, then it would make sense to spend resources to create a physically correct simulation of a fire. But for now, a simple model to simulate the fires suffices.

It was chosen to use a variation of conduction to increase the temperature of neighbouring objects around a fire in a pseudo-scientific way. The temperature increase by conduction is calculated with all the physical properties of the flammable object and the properties of the nearby fire (e.g. distance from the fire, temperature of the fire). The heat transfer that happens by conduction is really small. So, small that it could even be discarded in most cases. That is the reason why a multiplication factor is used. This multiplication factor is a constant that is determined through experimental testing to see which factor makes the fires spread between objects in the most realistic way. The temperature increase due to conductivity is multiplied by this factor. This class also updates the core temperature of the flammable object itself. This is updated in the same way as the temperature of the core.

5.3.2.3 FireSpreader

The *FireSpreader* script is attached to the *Fire* game object. This script will give the fire the ability to spread around in the scene. It handles both requests to spread fires between objects as well as requests to spread fires inside an object even though the two types of spreads are done differently. When spreading fires, the goal is to find the positions where new fires have to be spawned. Finding these points is rather tricky. It can be chosen to spawn fires at a fixed distance from each other without taking care of any other parameters, but that does not look good.

The approach that is implemented in the firefighting simulator is rather different. It makes use of *OverlapSpheres*. What an *OverlapSphere* does, is to create a sphere around a centre point with a predefined radius and return all the colliders that are inside or touching that sphere.

An *OverlapSphere* with a mask and with a radius that depends on the size of the fire is executed around the origin of the fire. Because a fire grows automatically with time, until it reaches its maximum size, the radius of the *OverlapSphere* will do the same. The mask used for the *OverlapSphere* will make sure that only the colliders of flammable objects are returned. Once the list with all the colliders is returned, a couple of colliders have to be removed. First, the collider of the flammable object on which the fire is burning is removed because this method will only be used to spawn fires on other objects. Then, every collider that was already found by a previous *OverlapSphere* is removed. All of the remaining colliders will be added to a list of colliders called *hitColliders* which is a field of the class. If the *hitColliders* list contains colliders that have not been found in a new search, they are also removed from the list. In the current version of the simulator this cannot occur because objects cannot be moved. But if the Navy wants to make the experience more realistic in the future by adding the feature that objects can be moved by spraying them with a powerful water jet, then this feature would come in handy.

Because an *OverlapSphere* is a resource expensive operation, it is only executed every time the size of the fire is increased. This size increase happens every second until the fire has reached its maximum size. This means that once the fire is full scale, no *OverlapSphere* is executed which saves resources. It is also important to know that every fire grows independently which signifies that they are not updated all at once but asynchronously. As a consequence, the *OverlapSphere* operation for every fire happens on a different moment in time which prevents fps drops. Figure 5.6

shows how an *OverlapSphere* looks when it is executed. When the sphere touches the box on the left, the box's collider is added to the *hitColliders* list.

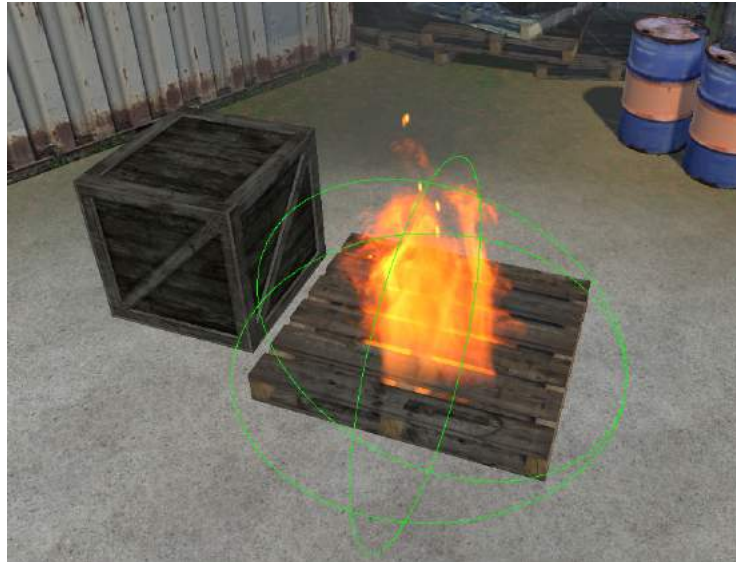


Figure 5.6: Fire with its *OverlapSphere* (green) in Unity

The next step is to find positions to create *FireCores* on the new colliders that have been added to the *hitColliders* list. To get these positions, the closest point on every new collider from the fire centre is calculated. A *FireCore* is created in those points and is added to the *fireCores* list. These *FireCores* will regulate the temperature in every position and will send out a request to spawn a fire once the ignition temperature for a certain object is reached in one of the cores. The temperatures of all the cores are updated every second thanks to a coroutine called *UpdateCores*. The coroutine will request the *FireSpawner* to spawn a fire once the ignition temperature for a certain core is reached. At that moment the *fireCore* object will be removed from the *fireCores* list and all references to the core are deleted. This means that during the next garbage collection, the memory taken up by the core will be freed up.

A *fireCore* is also removed from the *fireCores* list when it still has not ignited after a certain number of updates. An update happens every second, which means that if the number is 200 for example, every *fireCore* that has not ignited after 200 seconds will be removed from the list and the memory will be freed during the next garbage collection.

It is important to remember that a *fireCore* is deleted once it reached the ignition temperature and it ignites a fire. From that moment forward, it is the fire that will regulate the temperature in that position. Because every Fire object has a *FireSpreader* attached, this process is repeated for every new fire that is spawned.

5.3.2.4 FlammableObject

Every flammable object in the scene has this script attached. It is used as a sort of container for all the physical specifications of a certain object. These specifications are needed to spread fires

inside and between objects. The choice to make the fire spreading be dependent on the physical properties of an object is the main reason why this class is needed. This class will also change the tag and layer of an object to make the object detectable by the *OverlapSphere* of the *FireSpreader*. In the current implementation of the VR simulator, only class-A fires are implemented. This means that only solid objects have the *FlammableObject* script attached. This class is also responsible for the fire audio. There are three types of audio files used, the *Fire Place Low Intensity*, *Fire Place Medium Intensity* and *Fire Place High Intensity*. These audio clips are part of the *Fire Burning* package that was purchased from the Unity asset store. These three audio clips are played dependent on the health of the fire that is present on the burning object. If multiple fires are present on a single object, then the fire with the highest health will determine the audio clip that is played. It was chosen to only play one clip per burning object to limit the noise through the speakers of the HMD. The HTC Vive pro headset has low quality headphones, thus it can get really noisy, really fast. After some tests this way of working seemed to give the best user experience.

5.3.2.5 Fire

The *Fire* script is added to a *Fire* game object once it is instantiated. This script will control the behaviour of the fire. It is responsible to keep track of the fire's health, scale and temperature. The *GrowFire* method is the coroutine used to update the fire every second. This coroutine calls up several methods to increase the fire's health, scale and temperature. It will also lower the position of the fire if it is not at the lowest point of the AABB of the object.

The fire's health is the main parameter that controls both the scale and the temperature. If the health is changed, the two other parameters will also be influenced. Both parameters have a linear relationship with the health that is dictated by the formulas in Listing 5.4, lines 9 and 25.

The health is increased every second by the *IncreaseHeath* method. This method has two possible ways to increase the health of a fire. It can increase the health linearly by incrementing it or it can increase the health based on an addition with the *coreTemperature* that is multiplied by a multiplier. The first method is used in the *ClickUpdate* method that is called when a user clicks on a fire to increase its size. The second method is used in *GrowFire* which is called every frame.

```

1 private void UpdateScale()
2 {
3     float sizeX = fireCollider.bounds.size.x;
4     float sizeZ = fireCollider.bounds.size.z;
5     // Has to be edited if it has to work properly for big objects
6     if ((sizeX < maxSize) && (sizeZ < maxSize))
7     {
8         // Equation of the first degree
9         scale = scaleIncrement * health + (maxSize * Vector3.one - scaleIncrement *
10            maxHealth);
11         gameObject.transform.localScale = scale;
12         fireSpreader.UpdateFireCoresList();
13     }

```



```
13 // If the fire has reached its maximum size and the neighbours are not yet
    spawned
14 else if (!neighboursSpawned)
15 {
16     // This function is only called once by the fire script.
17     SpawnNeighbouringFires();
18 }
19 }
20
21 private void UpdateTemperature()
22 {
23     if(coreTemperature < maxBurningTemperature)
24     {
25         coreTemperature = health * temperatureIncrement;
26     }
27     Mathf.Clamp(coreTemperature, 0, maxBurningTemperature);
28 }
```

Listing 5.4: *UpdateScale* and *UpdateTemperature* method.

To update the scale of the fire, the *UpdateScale* method is used. This method will update the scale based on the formula on line 9 of Listing 5.4 until the fire reaches its maximum scale. The maximum scale of a fire depends on the size of the object that is burning. Once the maximum scale of the fire is reached, this method will call up the *SpawnNeighbouringFires* method.

In reality, the size of the fire does not only depend on the size of the flammable object (fuel). The amount of oxygen plays a big role in the propagation and lifetime of a fire. In a room where there is plenty of flammable objects, but where oxygen is limited, the fire goes out automatically. For this application, it is assumed that in each virtual environment, there is enough oxygen for the fires to grow until they reach their maximum size.



Figure 5.7: A fire with its possible spawn points (white) in Unity

The *SpawnNeighbouringFire* methods is responsible to spread fires inside an object. It takes a look at 26 possible spawn points around the current fire (Figure 5.7) and will look if a fire could be spawned in them. The criteria to spawn a fire in a spawn point is that the point is inside the bounds of the AABB of the burning object. If that is not the case, nothing happens. But if the spawn point is inside the bounds of the AABB, a call is made to the *FireSpreader* attached to the fire to create a *FireCore* in the specific spawn point. This *FireCore* will keep track of the temperature in the spawn point and will make the *FireSpreader* spawn a Fire once the temperature reached the ignition temperature of the already burning material.

The temperature of the *Fire* game object is updated using the *UpdateTemperature* method which will implement the formula from line 25 in Listing 5.4.

The last method executed in the *GrowFire* coroutine is the *LowerFireHeight* method which will lower the Fire game object's height if it is not at the bottom of the object. If a fire is somewhere at the top of an object, it will look like the fire is spreading to the bottom. The top part will still be engulfed in flames from the fire even though the fire has moved to the bottom. This is because the fire is increasing in size while it is lowered.

Finally, there is one last method in the *Fire* script and that is the *Damage* method. This method is called every time the fire is hit by a water particle. It will decrease the health of the fire like seen in Listing 5.5, line 5. It can be seen that the smaller the health the faster it will decrease. If the fire's health drops to zero, the *killMe* flag is raised to let the *FireManager* know that this *Fire* object can be destroyed.

```

1 public void Damage ()
2 {
3     // Damage fire
4     if(health > 0)
5     {
6         health = (int) (health - (1 - (health-1)/maxHealth) * HEALTHDECREMENT);
7         health = (int) Mathf.Clamp(health, 0, maxHealth);
8     }
9     // Destroy fire
10    if(health <= 0)
11    {
12        KillMe();
13    }
14 }

```

Listing 5.5: *Damage* method of the *Fire* class.

5.3.2.6 CameraScript

When placing fires with the computer or when observing a training session, the instructor has to be able to move around in the scene. This is achieved by adding the *CameraScript* to the *InstructorCamera*. This script will give the instructor the ability to move the camera using the arrow-keys and the mouse.

The arrow-keys will translate the camera in different directions in the scene. Using the mouse and the middle mouse button, the user can rotate the camera in different directions.

5.3.2.7 DestroySmoke

Before a *Fire* game object is destroyed in the *FireManager*, its child object *Smoke* is detached from it, so it could create the effect of smoke lingering around after the fire has been extinguished. Eventually this smoke will also have to disappear. This is achieved by making the emission rate of the *Smoke* particle system zero when the fire is extinguished. Doing this will ensure that no new *Smoke* particles will be emitted. This combined with the fact that *Smoke* particles have a limited lifespan will make the *Smoke* particles disappear after a while. Even though no smoke particles will be visible, the *Smoke* object will still be in the scene because it is never destroyed. This means that smoke is taking up memory without even being used. During a training session these *Smoke* objects will start to pile up and they will start taking up more and more space which will deplete more system resources and eventually have an impact on the user experience.

To avoid this from happening, the *DestroySmoke* script is attached to the *Smoke* game object. This script will automatically destroy the smoke game object once its emission is zero and all the particles from that object have disappeared.

5.3.2.8 GameManager

After the instructor is done placing all the fires, a button needs to be pressed in order to start the next phase of the game. When using the keyboard, this is the *TAB* key. When using the controller, this is the *Menu* button. When one of these buttons is pressed, the laser pointer is deactivated. The *UDPServer* is activated and starts receiving packets from the extinguisher. After processing the first packet, the server spawns the extinguisher into the scene.

This script is also responsible for ending the game when all fires are extinguished. A coroutine checks every ten seconds the number of fires that are still burning. When the amount of fires is zero the Boolean *endGame* is set true. If, the next time this coroutine is executed, the amount of fires is still zero, the game is ended and the *EndMenu* screen is loaded.

5.3.3 Implementation of the MR simulator

5.3.3.1 MRFire

The *MRFire* script is a simplified version of the *Fire* script. But the *Fire* prefab is the same as described in section 5.3.1.6. This fire cannot spread or spawn new fires, because in order to do so, the simulator should have a notion of different objects. This is not the case, since it only overlays virtual objects on live camera images.

The script has three public methods, *SetHealth*, *IncreaseHealth* and *Damage*. The first method, *SetHealth*, is called when the fire is instantiated. It sets the health of the fire to a default value. The second method, *IncreaseHealth*, increases the health with the same default value. The scale

changes accordingly. The third and last method, *Damage*, decreases the health with one unit. The scale changes accordingly also.

5.3.3.2 Laser pointer

The instructor needs to be able to place fires. This can be done using the default VIVE controller as a laser pointer (Figure 5.8). This laser pointer is created using different pre-made and custom scripts. In the scene, an empty game object, called *LaserPointer*, is created. Three scripts are added to this game object: *SteamVR.Behaviour.Pose*, *ZEDControllerTracker.DemoInputs* and *SteamVR.LaserPointer*. The laser pointer game object has one child game object called *FirePlacement*. This game object has only one component, the *MRFirePlacement* script.

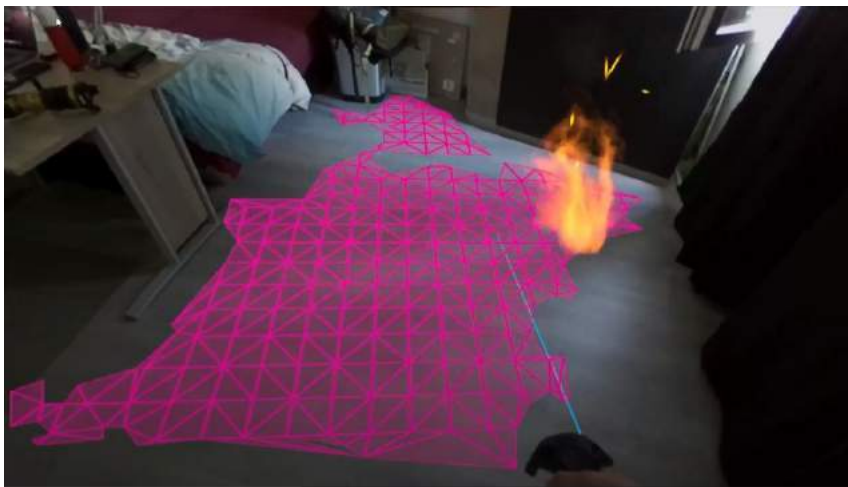


Figure 5.8: MR simulator game view with laser pointer (blue) and plane detection (pink)

Players might have different preferences concerning the position of the controller in their hand, so the position of the pose can be set in the bindings UI. A pose represents the position and rotation of a tracked object. The *SteamVR.Behaviour.Pose* script gets the position and rotation of a certain controller. (Valve Corporation, 2019) The action pose is set to SteamVR's default pose of the right controller.

The *ZEDControllerTracker.DemoInputs* script tracks a certain device, in this case the right controller. The script allows to set actions that should be executed when a certain event, like a button press, occurs. These events are set using the bindings UI. This script requires to set a *ZEDManager*. In this case, the *ZEDManager* of the *ZED_Rig_Stereo* is used. A *ZEDManager* manages a ZED camera. It is responsible for setting the right parameters and opening and closing the connection with the camera. (Stereolabs Inc., 2020)

The *SteamVR.LaserPointer* script adds a virtual laser beam to the controller. In contrast to the VR simulator, no 3D model of the controller is needed since the user can see the real world and thus the controller. Three functions, for three different events, can be implemented by the developer. For this laser pointer, a function for only one event, *PointerClick*, is needed. This script requires an action, which causes the *OnPointerClick* function to be called. In this case, the *InteractWithUI*

action, which is bound to the trigger, is used. The function is added to the laser pointer by the *MRFirePlacement* script.

Inside of the *OnPointerClick* function, the target of the laser is verified. If the laser did not hit anything, nothing happens. If it hit a *ZEDPlaneGameObject*, the normal of the plane is retrieved. If the plane is close to horizontal, *SpawnFire* is called. This function spawns a new fire at the hit point of the laser pointer. It instantiates a new instance of the fire prefab and adds the *MRFire* script. The health of the fire is set to a default value. If the plane is not close to horizontal, nothing happens. If an existing fire is hit, the fire's health and size increase. This is possible until it reaches a maximum value. A fire with a higher health needs to be hit by more water particles before it goes out. To determine if the hit object is a fire or a plane, the target's tag is verified.

With the default laser pointer script from SteamVR it is not possible to get the exact position of the hit, only some information about the target object. That is why the scripts were modified. The *RaycastHit* is made public. The *RaycastHit* has different attributes like the collider that was hit and the impact point of the hit.

To detect planes, the *ZEDPlaneDetectionManager* script is used. On calling the *DetectPlanes* method, the plane detection manager starts detecting planes around the given location. The plane detection manager detects flat surfaces and adds them to a list. These detected planes are highlighted in pink in the scene and game view, when this option is selected. In this simulator, this is only the case during the first phase of the game, when the instructor is placing the fires.

5.3.3.3 Other functions

When the instructor is done placing fires, he can start the next phase of the game, which is extinguishing the fires. This can be done by pressing the Menu button on the controller. When this button is pressed, the laser pointer is disabled and disappears. The *FireSpawner* script is also deactivated, which means that no new fires can be added. The *ZEDPlaneDetectionManager* keeps detecting planes, but it will no longer highlight them in the game view. The last thing that happens is the activation of the *UDPServer*. The server will start receiving UDP packets from the extinguisher and adds its prefab to the scene.

Next, the trainee can start extinguishing the fires. To do this, one of the available extinguishers is used. After booting the extinguishing device, it will start sending UDP packets. These packets are received and read by the server that runs inside of the application. The *AgentInteraction* script processes this data and translates it to changes in the shape and flow rate of the water beam.

5.4 Object detection

Fires in the VR application are able to spread from one object to another in a pseudo-physical manner, taking into account different parameters, like the material. In MR, the fires are overlaid on live video footage. There is no notion of different objects. The fires can increase in size, but they will not spread from one object to another. In order to spread from object to object, single objects need to be recognized from the live video footage. Since the spreading of fire depends on

parameters, like the material, the MR application needs to recognize this too. Machine learning and object detection can be used for this. This thesis focusses on the detection of materials. Due to limited time, this part could not be fully implemented.

In the field of computer vision, object detection is a technology that not only recognizes a certain object, but also finds its location in the image. (MathWorks, nd) It indicates the location of the object by drawing a bounding box or a mask. A bounding box is the smallest area possible, following the axes, that contains the object. A mask indicates the location of the object following its contours.

For this thesis, the TensorFlow Object Detection API is used. This open source API allows users to easily build, train and use object detection models. The API comes with a bunch of pretrained models that can be deployed immediately. It also includes everything needed to train an existing model on a custom database. (TensorFlow, 2020a)

Most pretrained models in TensorFlow's API are trained on the COCO dataset from Microsoft. The COCO dataset, or Common Object in Context dataset, consists of more than 300 000 images that belong to 80 classes. Images in the dataset can contain multiple objects from different classes. Compared to other object recognition datasets, COCO focusses on detecting single instances of classes. (Lin et al., 2014)

5.4.1 Dataset

First, a suitable dataset needs to be found. For this project, the Flickr Material Database is used. This dataset consists of 1000 images of 10 different classes. Each class consists of 100 images, from which 50 are regular views and 50 are close ups. The images were manually selected by the creators of the dataset to ensure the diverseness in lighting conditions and colours of the dataset (Sharan et al., 2010)

In order to train a TensorFlow model on a custom dataset, TFRecord files need to be created. A TFRecord file is a serialized version of the data. It stores a sequence of binary records which makes it easier to read for the model. (TensorFlow, 2020b)

The data is split in a training set and an evaluation set. Next, a TFRecord file is created of both sets. To do this, a python script is written that splits the data evenly. From each class, ten close-ups and ten regular views are put in the evaluation set. All other images are put in the training set. The evaluation set now contains 20 images from each class. All images are cropped to 300*300 pixels instead of 512*384.

A label map needs to be created. A label map file binds an id, starting from one, to a class. Before the images can be converted, each class is represented by its id.

5.4.2 Model

Second, an algorithm needs to be found. Speed was the most important factor when choosing the algorithm, since the frame rate had to be at least 90 fps. If the frame rate drops far below 90 fps, the user will start to feel nauseous and the MR simulator becomes unplayable. This is why a single shot detector model was chosen. SSD models are the fastest kind of models available in the

TensorFlow Object Detection API and they are designed for real-time object detection.

SSD models take a single image or video frame as input. It outputs the probability and location of detected objects. SSD models consist of two major parts: the backbone and the SSD head. The backbone is a pretrained convolutional neural network for image classification. In an SSD model, it is used as a feature extractor. The backbone is extended with the SSD head. The SSD head consists of one or more convolutional layers. Its outputs are the bounding boxes and classes of detected objects. (ArcGIS API for Python, nd)

SSD uses a set of default bounding boxes and tries to predict the class and offset for each box. This significantly increases the frame rate of the model. Different scales of feature maps are used to obtain a high accuracy. A feature map is obtained by dividing the image in cells using a grid. Different scales of grids result in different feature maps. A small number of boxes, which have different aspect ratios, is put at each cell of a feature map with different scales. These boxes go beyond the boundaries of the cell. For each box, the offset and confidence of each class is predicted. (Liu et al., 2016)

5.4.3 Training

Before the model can be trained, a pipeline needs to be configured. This config-file consists of five major parts:

- *Model*: this defines the model that will be trained
- *Train_config*: in this part, the parameters to train the model are defined
- *Eval_config*: this defines the metrics that will be used during evaluation of the model
- *Train_input_reader*: this defines the data for training (TFRecord file)
- *Eval_input_reader*: this defines the data for evaluation (TFRecord file)

The config file of the pre-trained model, with some changes, can be used for this. First of all, the number of classes needs to be changed to the number of classes in the dataset. Second of all, the user has to change the path of both the training input and evaluation input file and the path to the label map file. Next, the number of samples in the evaluation set needs to be changed. The user can also define a checkpoint file. This file also come with the pre-trained model and speed up the training process. (TensorFlow, 2018)

After configuring the pipeline, training can start. TensorFlow provides a *train.py* script that starts the training process. When training is finished, the trained model can be exported using the checkpoint files. Multiple files are generated from which the *frozen_inference_graph.pb* contains the actual model and is used to deploy the model. Due to time constraints and an unresolvable error during training, this part and the part described in section 5.4.4 of the project could not be completed.

5.4.4 Object detection with ZED

Stereolabs provides a python script that applies object detection to the live video feed from a ZED camera. In the script, the path should be changed to the path of the new trained model. When the script is executed, it opens a new window on the screen which shows the processed video footage (Figure 5.9). Bounding boxes are traced around the detected objects, its predicted class and probability are shown and the distance to the object is measured. The low frame rate should be taken into account.

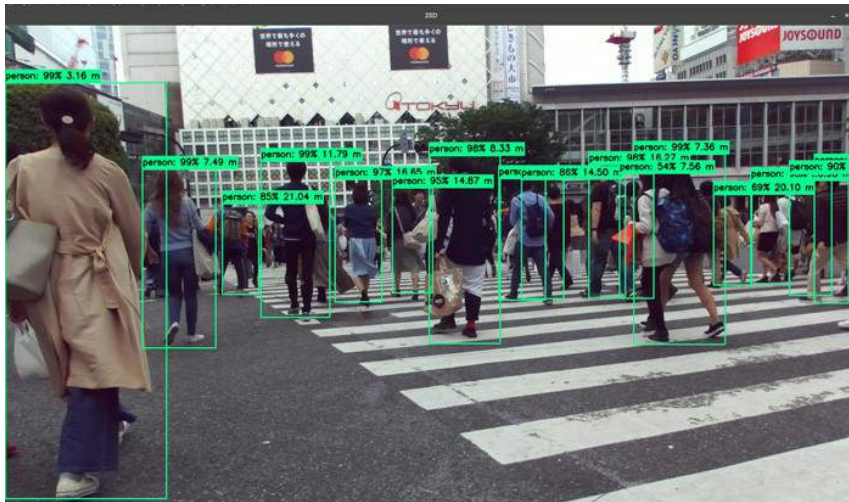


Figure 5.9: Object detection with a ZED camera (Stereolabs Inc., ndb)

With Python for Unity, python scripts and Unity can work together. The live video from the ZED Mini should be processed by the script in order to be able to get the kind of material of objects in the room. This information can be used by Unity to spread fires accordingly. Take into account that object detection requires a lot of processing power. Using Stereolabs' python script with a pre-trained model the frame rate drops to 30 fps. This is even without the processing power that is needed for Unity to run smoothly. When object detection would be used in combination with the MR simulator, it would become unplayable.

When working with the ZED 2 camera, object detection can be enabled directly in Unity. This would make it a lot easier to implement. An external python script would not be needed. Unfortunately, this camera was not available and is not designed to be attached to a headset.

Chapter 6

Evaluation

6.1 Performance

To test the performance of the system, Unity's build-in profiler is used. Unity's profiler allows developers to verify which task, for example rendering or garbage collection, uses the most resources at a certain time. It also indicates the frames per second. For an XR application, a refresh rate of at least 90fps is recommended otherwise, the user can start to feel nauseous. This is called simulation sickness. (One Bonsai, nd)

Unity's profiler indicates the frame rate, but some considerations should be made. The indication of the frames per second is not very reliable and exact, but it can be used to get a general idea about the differences between two cases. It should also be taken into account that the profiler is used in the editor, so before the project is built. This causes some overhead, but it can be generally accepted that after the project is built, the frame rate would improve.

6.1.1 VR simulator

Figure 6.1 shows a screen capture of Unity's profiler while the VR simulation is running. It was taken right after 15 fires were spawned, so they were still very small. The average frame rate is around 90 fps. The peaks in the CPU usage are because of Unity's garbage collector.

Figure 6.2 also shows a screen capture of Unity's profiler, but a minute later. Those 15 little fires have spread and caused new fires to spawn. These new fires also cause new fires to spawn and so on. The frame rate dropped drastically to 30 fps.

Comparing both views of the profiler during the VR simulator shows that the frame rate drops drastically when many particle systems are used. The second screen capture would make you think the simulator is unplayable, but that is not true. When trying on the headset, the simulator appears to be perfectly fine. This could be the fault of the profiler itself. It is known for giving false values for the frame rate.

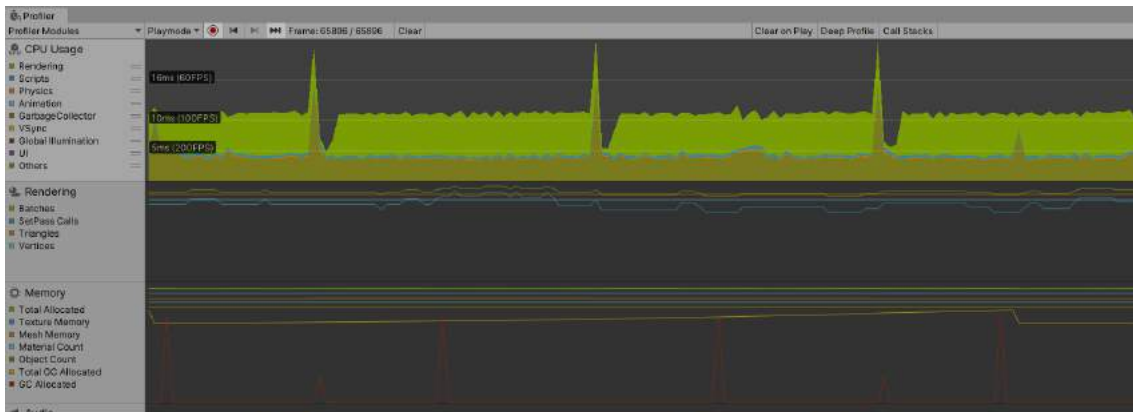


Figure 6.1: Unity profiler during VR simulation with 15 small fires present



Figure 6.2: Unity profiler during VR simulation with more than 15 fires present

6.1.2 MR simulator

Figure 6.3 is a screen capture of Unity's profiler while the MR simulation was running without any fires present. The average frame rate is just above 60 fps, which is actually too slow for the user to be fully immersed in the simulation. Again, peaks in the garbage collection can be noticed.

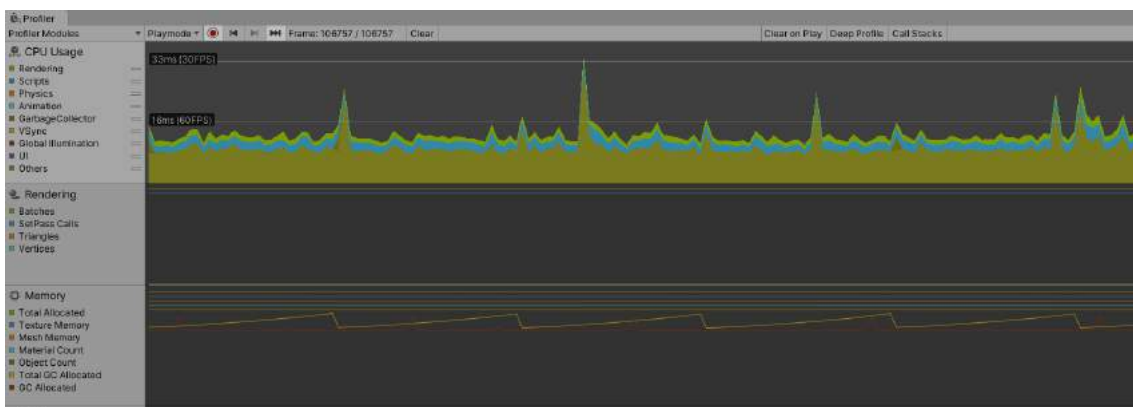


Figure 6.3: Unity profiler during MR simulation with no fire present

Figure 6.4 is also a screen capture of Unity's profiler while the MR simulation is running, but this time there is one fire present in the middle of the room. From the image, it can be derived that the average frame rate is around 90 fps. This should be fast enough to play the simulator comfortably.

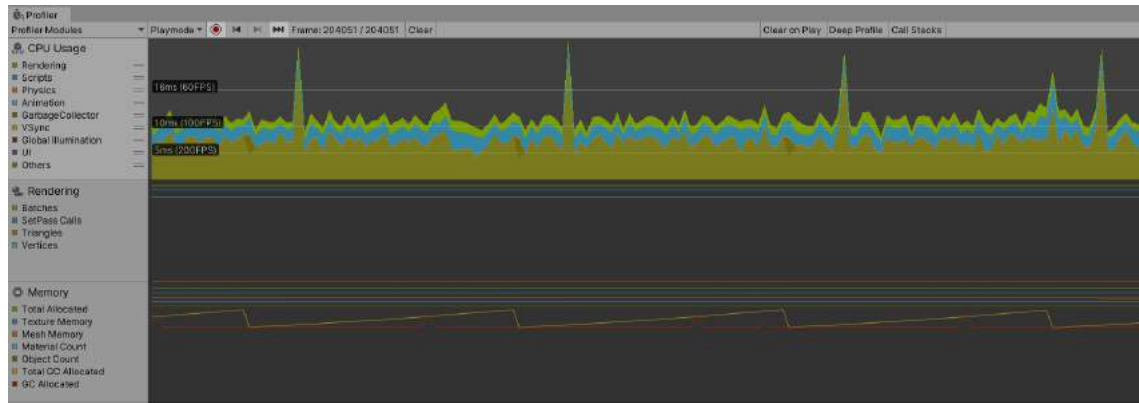


Figure 6.4: Unity profiler during MR simulation with one fire present

When both screen captures of the profiler during the MR simulation are compared, some strange results can be noticed. The frame rate of the simulation with a fire present is higher than the frame rate of the same simulation without a fire. This is the opposite of what was expected since a particle system requires a lot of resources and a fire consists of four particle systems. This strange result could be due to a wrong indication of the profiler. If the MR simulator is compared to its VR counterpart, it can be seen that a lot more resources are needed for animation in the MR simulator. Rendering, on the other hand, uses fewer resources.

6.2 Durability

For the firehose nozzle, multiple parts were 3D printed. Their robustness should also be evaluated. The parts were printed in PLA. After each part was printed, its firmness was tested by pulling the features that could be a point of weakness. When the part broke, it was redesigned.

The first part that had to be redesigned is the backplate with insert. In the first few prototypes, the backplate and its insert were one part, but the insert kept breaking off like seen in Figure 6.5. The design was changed so the actual backplate and the insert would be two separate parts. This made it more robust and easier to replace if one part would break again in the future.

The second part that had to be redesigned is the Arduino housing. The Arduino sits about 5 mm above the floor of the container. The first few prototypes had little pins on the four corners that would fit in the holes of the Arduino. It turned out that these pins broke off easily. For that reason, a thicker base was added to the pins. But even these bases would break off from time to time (seen on the left in Figure 6.6). The right circle on Figure 6.6 shows a thick base, but with a broken pin on top. The final design of the Arduino housing has two raised platforms and walls on each side to hold the Arduino in place. This is a lot sturdier and does not brake of that easily. Velcro was used to attach the Arduino to the platforms.

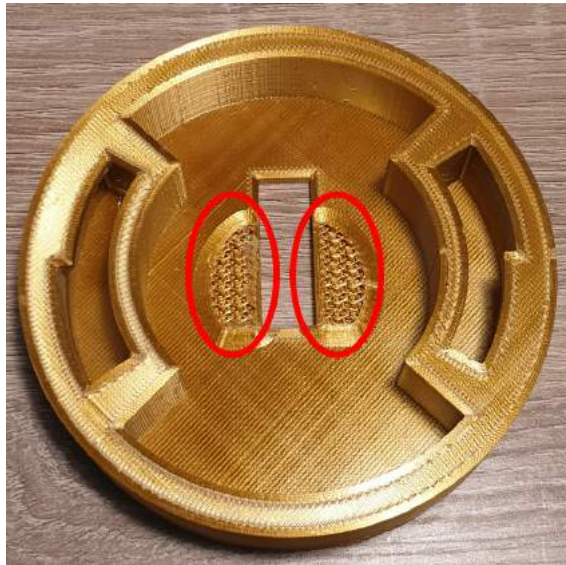


Figure 6.5: Broken backplate (early version)

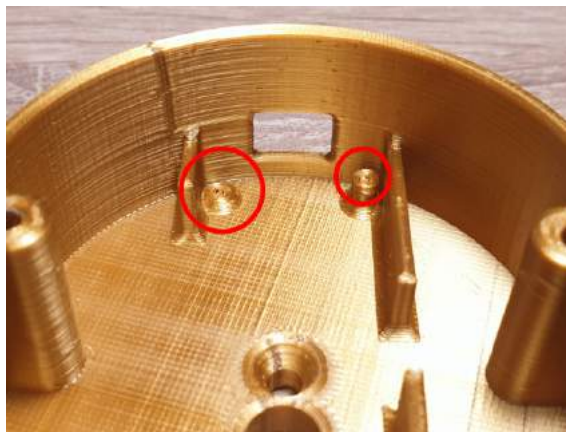


Figure 6.6: Broken pins in Arduino housing (early version)

In section 4.1.1 it is mentioned that a small part of the base of Figure 4.3 is cut out to be glued on the inside of the nozzle and prevent the head from rotating. It is the only piece together with the firmly tightened screw that keep the head in the right orientation. Unfortunately, even if the screw is fully tightened, the head can still be rotated when a decent amount of force is applied. This can happen when one accidentally grabs the head instead of one of the selector rings. The fact that the head can still be rotated is due to a bad adhesion of the printed piece to the metal part of the nozzle. The piece is able to move around which defies its purpose. Using another type of glue might offer a solution. Gluing the whole base to the inside of the nozzle might also help. But the best way to solve this problem is to redesign the base of Figure 4.3.

Chapter 7

Discussion

In this chapter a critical reflection is made of the whole system. The simulator is compared to other existing systems like those mentioned in section 2.2 and we take a look to which degree the requirements of section 3.2 and 3.3 are met.

7.1 General considerations

The differences between VR and MR make it difficult to compare the two simulators directly with each other. Both technologies have different purposes and use cases, and they both have their strengths and weaknesses. A lot more things can be simulated in the virtual environment because of the fact that the environment does not depend on any real-world phenomena like light or reflections. There are fewer constraints in VR than in MR. In MR, special care has to be taken with the objects present in the simulation environment, the lighting in the environment and reflective surfaces. Both bad lighting and reflective surfaces can confuse the ZED camera which will result in a malfunction when the user wants to spawn a fire.

Reflective surfaces can also interfere with the VR tracking. It can cause the tracking to malfunction which will result in glitches in the simulation that can cause discomfort and may reduce a player's performance. In the case of VR, the interaction between the user and the simulation happens in a complete virtual environment thus the user will not have to interact with reflective surfaces like they would have to when playing in MR. This means that reflective surfaces will not hamper a user's ability to spawn fires in VR like it would in MR.

Because the MR simulator also uses the VR tracking to track virtual objects in 3D space, it will also be affected by all the drawbacks caused by reflective surfaces. In this case, the virtual objects in the scene will start glitching. This is less pronounced than what happens in VR, where the whole view will start glitching.

The firehose nozzle will also be prone to glitches because it is tracked with the same technology as the HMD, namely inside-out tracking via SteamVR Tracking 2.0.

Another external factor that should be taken into account is the heavy machinery on board of a ship that cause a lot of vibrations. These vibrations have an effect on the whole ship and thus, no matter

where the Base Stations 2.0 are attached, they too will vibrate. This will disturb the tracking and will affect the user experience.

Since a ship moves with the swell, even when it is docked in a harbour, these constant movements might cause some discomfort for the user when training in a fully immersed virtual environment.

So, in theory the system would be very useful on board and offers a lot of advantages. But when all external factors are taken into account, it can be concluded that further research and development is needed to make the XR technology suitable for use on board of a ship.

7.2 Evaluation of the Navy's requirements

To determine whether the simulator meets the needs of the Belgian Navy, each requirement is evaluated. This gives a better view of whether this project succeeded or not.

7.2.1 Evaluation of functional requirements

The first functional requirement is the use of immersive technologies to train the crew. The simulator should be convincing enough, so it actually adds value to the training procedure. Due to the Corona protective measures taken by the government, no user tests could be done. These user tests were supposed to provide feedback on whether the simulator is realistic enough. The real-life tests were replaced with videos that were evaluated. But relatives and roommates could be counted on to test the system in person. Most of the feedback that was given, was taken into account but due to limited computing power and time constraints, some remarks could not be implemented.

The second functional requirement is that the MR system should be compatible with extra props and equipment they already possess. This is also the reason why live plane detection was used instead of scanning the room before starting the simulation. A mesh can be made from the room, but when a prop like a door is used, the system has no idea of what is behind the camera. Using live plane detection ensures that this requirement is met. The simulator should also be playable when wearing a firefighting suit and gloves. By eliminating wires and routing them so they cannot be reached during gameplay the robustness of the nozzle is increased. No wires can accidentally be pulled.

The third requirement states that the instructor should be able to place fires and that after the fires are spawned, they spread on their own. The fires are placed using an input device, a VIVE wand or keyboard and mouse. In VR, the *FireSpreader* script makes the fires spread from one object to another. In MR, this is not yet implemented, since no different objects can be recognized yet. To do this, a machine learning algorithm should be implemented, but will make the frame rate drop and the simulator will become unplayable.

The fourth requirement is the use of a device, that resembles the real-life firehose nozzle, to extinguish fires. This is done by using the same firehose nozzle that is also used during hot training. The model is equipped with sensors to track the different manipulations and to visualize them accordingly in VR. In MR, only changes to the water beam are made. Also, no virtual model of the nozzle is used in MR since the player can see the real-life nozzle.

7.2.2 Evaluation of non-functional requirements

The first non-functional requirement covers the mobility of the system. The whole system should be easy to move around. This should be analysed by looking at the different components of the system. First, the HTC VIVE Pro kit is needed. This kit contains the Vive pro headset, Vive pro controllers, Linkbox and the SteamVR base stations 2.0. Everything is easy to set up, but extra thought should be put in the base stations. The base stations should be placed at opposite corners of the room and they need access to an electrical outlet. The base stations should be put at a height of around 2 m, so holders should be stuck to the wall or the base stations should be put on a tripod. This makes it more difficult to move around. The other parts of the system are the backpack pc, firehose nozzle and the router. The router should only be connected with its power cord, no ethernet connection is needed. The pc and nozzle are very portable. Due to the nature of the SteamVR Tracking 2.0 principle, more thought should be put on the setup of the system which makes the whole system less portable. This is a weakness of the system but cannot be avoided when using the HTC VIVE Pro kit.

The second non-functional requirement states that the system should be easy to set up. This was also going to be a part of the user tests, so no feedback on this part was received. The SteamVR system is fairly easy to set up. Most of it consists of plugging in devices which are detected automatically by the computer and afterwards a small SteamVR setup tutorial must be followed to calibrate the HTC Vive pro setup. The firehose nozzle only needs to be connected to the battery and the switch on the back should be switched on. Overall, the system is easy to setup. The third and last non-functional requirement covers the extensibility of the system. It should be easy to add new features, like other fire extinguishing devices such as CO_2 -extinguishers, powder extinguishers, different type of water hoses, etc. The code is written with this in mind. To test this, a new extinguisher was added to the system. It did not take longer than 15 minutes to implement it, when a proper prefab is used.

7.3 Comparison with related words

If we compare our system with the analysed systems in section 2.2, we can see that there are some similarities but several differences between our systems.

7.3.1 Comparison with the FLAIM Trainer

What FLAIM systems wanted to realise with the FLAIM trainer is the development of an immersive VR fire training simulator. They emphasised the immersion of their system by highlighting all their accessories like the heat vest, the mask, the hose-line system and the firehose nozzle itself. Their main form of interaction with the system is the FLAIM firehose nozzle that is attached to the hose-line system.

If we compare our two systems, we can see that our main focus was creating a system that is intuitive, immersive and realistic. If we look at the main form of interaction, we can see that it is

almost the same. We both use a firehose nozzle as a controller to interact with the system. The nozzles are both tracked with a VIVE tracker and they almost have the same controls.

But there are some major differences. The firehose nozzle that we use, is the exact model that the Belgian Navy uses, and which trainees will have to work with. This means that our controller feels and acts like the original AWG TURBO 2230. Users will not have to learn to work with a new type of nozzle, they will be able to interact with the system in the same way as interacting with the real firehose nozzle. This means that everything trained in the simulator can be directly conveyed to the real world without having to alter or relearn manipulations. Another added benefit of using the real firehose nozzle as the controller is that the controller is not fully made of plastic which means it is more robust and it has the right weight to it.

A drawback of our system compared to the FLAIM trainer is that the controller is the only haptic feedback we currently have to offer. The FLAIM trainer has multiple ways of giving feedback to the user, like the heat vest and the hose-line system. These systems give an extra level of immersion when they work properly. If they fail to work correctly, they can add an extra distraction which will take the user out of the immersion. Although these features are not present in our current system these can always be added in the future. This will be discussed in chapter 8, Future work.

Another drawback of our system is the graphics and realism of the simulation. Because of our limited resources, we only used free assets of the Unity asset store. This means that we were restricted in the environments we could create, unless we modelled the objects ourselves. FLAIM has a whole team of designers that can design specific objects and scenarios for their system. With the word scenario, we not only mean the virtual environments that they offer, but also the types of fires. Our system only offers class A fires, while from what we have seen, their system offers, class A, B and C fires.

Currently, we only offer one virtual scenario, but the benefit of our system is that the user can train in a real world setting without having to model all the objects around him. This is thanks to the MR simulator that is integrated into the system. With this system, each crew member can be trained in any compartment of any ship without the need for major changes to the system itself. In case of a change to the layout of a room in a ship or the arrival of a new ship, there is no need to design a new room.

Our fire simulations are based on approximations to save resources, while their simulations are based on the real physical equations. This means that our fire interaction is just an approximation. However, the goal of our application is not to make digital simulations of fire that approach reality and are based on real data. The goal of this Master's thesis is to develop an application that can be used to train people. This means that the propagation of fire should only look realistic. The use of an approximation of reality could also be an advantage, since the propagation of fire can be much quicker. One of the reasons why the Navy does not like to use the FLAIM trainer is that it approaches reality too much. The propagation of fire is fairly slow, which means that training one student takes too long, which makes it too expensive to use. In our simulation, this process is sped up. Trainees can practice the procedures and the supervisors can get a sense of the attitude of a trainee towards fire, with a less expensive tool. Another advantage is that less time is needed from the instructors, so other work can be done too. This becomes really relevant in times when there is

a lack of 'personal resources'.

7.3.2 Comparison with the fire trainer by B. Schlager

When comparing our two systems it is clear that her system is scientifically correct while ours is an approximation. She was able to simulate the different types of heat transfer by using the heat transfer framework of her research institute. This allowed her to solve the complex equations for radiation and convection while maintaining a descent framerate.

Her system had another type of controller, but it is also based on a Reality Based Interaction (RBI) like ours. She attached a VIVE controller to an extinguisher to give the user the feeling he was manipulating a real extinguisher. Which is exactly what we did, but with a firehose nozzle. These types of controllers improve the user's immersion.

Other than the points mentioned above, the systems look the same. Both systems were made to be easily extensible in the future and with user immersion as the primary goal.

7.4 General discussion

The final system is the product of five months of full-time work. This proves that in a few months, it is possible to develop a basic fire training system starting from a list of requirements. Which implies that a professionally made, fully-fledged system with multiple custom scenarios could be developed and deployed within a year. If only one or two scenarios are required, the system should be deployable within a couple of months.

The developed system is not only useful in a military environment but could also be deployed in the civilian world. Firefighters could safely train for tricky situations, which do not occur very often. It can even be used to train civilians for basic firefighting, so they can protect themselves when a small house fire occurs. The system makes it possible to train a lot of people in a short amount of time. This makes such a system ideal to be used in schools. Students can learn about the basics of firefighting and learn how to use extinguishers, which can be a live-saving skill.

The system is designed to be modular and expandable. By only switching out the controller, a whole new set of skills can be trained. Depending of the audience, one controller might be preferred over another. To train firefighters, a nozzle is probably the preferred extinguishing device, since they use it in their daily job. To train civilians, learning how to use a fire extinguisher is probably more useful, since these are the extinguishing devices that are widely available and present in public areas as well as in people's cars.

Chapter 8

Future work

Even though the simulators that were developed for this Master's thesis are usable, there is always room for improvement. While some elements of our work could be improved with newer and better technology, other elements could be readily implemented. The system could also be extended with additional features and functionality.

8.1 Improvements

The current simulator comes with a router that is configured with an SSID, password and static IPs. These parameters are also configured in the Arduino code. This system could be improved by letting the Arduino broadcast a packet on boot up. This packet contains a key that the server must recognize. The UDP server running inside the application will check the received packet and the key it contains. If the key is recognized the server will send a reply packet which contains its IP address in the header and a confirmation code in the payload. After verifying the confirmation code, the Arduino can configure itself so it will only communicate with the UDP server in the future. It is almost like the three-way handshake of a TCP connection, but only for the first packet. No static IP addresses need to be configured in the router anymore.

The measuring system of the firehose nozzle could be another possible improvement, more specifically the measuring system of the big lever. In the current system, this is done by some LEDs and an LDR. After some testing, it turned out that this measuring method is not very accurate and reliable. With the right tools, a potentiometer could be used to fulfil the same task. A hole should be drilled on the side of the nozzle, between the lever and flow rate selector. This hole is needed to route the wires from the potentiometer to the Arduino at the inside of the firehose nozzle (Figure 8.1).

Another solution is opening the firehose nozzle and integrating all sensors at the inside so no external parts (except from the VIVE Tracker) are needed. This method was also tested and resulted in a broken nozzle. Without professional tools it is not possible to open the nozzle which is to be expected because it was designed and build to withhold a huge amount of pressure. The Belgian Defence could also work together with AWG, the manufacturer of the nozzle, to develop a custom

nozzle with integrated sensors. This would greatly improve the robustness of the device. If building a custom nozzle is not possible and external integration is needed, these parts could be 3D printed in metal. Right now, these parts are 3D printed in PLA which is not very sturdy. Printing them in metal or another sturdy material prevents it from breaking when it falls on the ground.



Figure 8.1: Firehose nozzle with indication of position for potential hole.

8.2 Future technology

The ZED Mini stereo camera limits the quality of the MR simulator due to its low frame rate and low resolution. The limited bandwidth of USB 3.0 is the root cause. Now the ZED Mini has a refresh rate of 60 fps and a resolution of 720p. The limited field of view takes away part of the experience. The user cannot see what is happening in the corner of his eyes and this does not feel natural. The refresh rate of the actual MR simulator frequently drops below 60 fps. This is due to the processing of the images and rendering require a lot of processing power. This could be resolved by better, smaller and more power efficient GPUs, which do not exist yet. The reason why they have to be small and power efficient is because a backpack PC is used. This computer is fairly small as compared to a normal desktop. The backpack PC can be powered by batteries, but when the GPU is not energy efficient enough, the batteries will die far too quickly. The object detection with the ZED Mini is also limited to 30 fps which is way too low to use in combination with a headset. More powerful electronics could also solve this problem.

8.3 Suggestions and possible extensions

The simulator that is delivered at the end of this thesis is rather basic. There is only one class of fire implemented. Different classes of fires need to be extinguished in a different way and with a different extinguisher. To be able to train these scenarios too, multiple classes of fires and different

types of extinguishers need to be implemented. Multiple extinguishers can easily be added to the simulator since the code is written with this purpose in mind. The interaction between different classes of fires and the different types of extinguishers should be visually correct. When a fire is treated with the right kind of extinguisher, the fire should go out nicely. But if a fire is treated with the wrong kind of extinguisher, the consequences (e.g. a fireball when burning oil is extinguished with water) should be programmed.

The VR simulator only has one scene which does not resemble a typical training environment of the Navy. In the future, more and custom scenes can be added that resemble the environment in which the Navy is used to train, like specific compartments of a Belgian Navy ship.

Right now, the simulator can only be used by one player. Multiplayer is not yet implemented. This feature could make the simulator a lot more useful, since some exercises are done in teams of two or three. When using multiplayer in VR each player should be aware of the position of the other players. During some exercises, trainees change roles in the middle of the exercise, so the extinguishing devices cannot be bound to a certain player. They have to be interchangeable between players.

Like the FLAIM trainer, haptic feedback would also be a nice feature to add to the simulator. This creates a whole new level of immersion for the user. A heat suit lets the user know when he is close to a fire by heating up the suit. This would be useful when the user cannot see the fire due to the thick smoke in the room. A breathing mask can help users to get used to breathing with an SCBA but can also serve to measure their oxygen intake. Knowing the oxygen intake is a nice feature that can be included into an after-action report. Together with the amount of water used during a training scenario.

Chapter 9

Conclusion

The Belgian Navy is looking for a way to improve the firefighting training of its crewmembers. The current training method is dangerous, expensive and environmentally unfriendly. This thesis re-searches the possibility of using immersive technologies within the training program. A VR and MR simulator were developed in Unity using the HTC VIVE Pro kit and ZED Mini stereo camera. The main input device for both simulators is a firehose nozzle. An AWG TURBO 2230 nozzle was modified and integrated into the XR system. This model of nozzle is chosen because it is universally used by the Belgian Navy. This makes it easier for trainees to operate the system without learning new controls.

In both simulators, the instructor is able to place fires using a VIVE wand or a computer mouse and keyboard. After the fires are placed by the instructor, they grow and start to spread on their own. Fires can spread within an object and between objects. The spreading mechanism is based on pseudo-physical principles. The fires can be extinguished using the firehose nozzle which is equipped with sensors and a VIVE Tracker. The data from the sensors is sent over Wi-Fi to the server that runs in the application. The protocol that is used to send the data is UDP.

The VR simulator consists of one training environment, but other environments can be added in the future. By altering the number of fires and placing them on different objects, different training scenarios can be created.

With the MR simulator, it is possible to place virtual fires in the real world. Fires do not spread in or between object because the simulator has no notion of different objects. In order to achieve this, object detection should be implemented. The algorithm should also be able to recognize the material of the objects since this has an effect on how the fire spreads.

At the end of this thesis, a playable system is delivered. Its performance was tested and evaluated. Unfortunately, no user tests were performed due to COVID-19. But an alternative way of evaluating was performed.

A comparison was made between the developed system and other systems or researches. The whole system can be extended by adding a handful of features, for example, other extinguishers, multiple classes of fires, more training scenarios and the ability to play in multiplayer.

XR is an interesting technology for the Belgian Navy. It has numerous benefits compared to con-

ventional training, like improved safety, reduced costs, more training scenarios and environmental benefits. So, it can be concluded that the Navy should adopt XR and other immersive technologies for training in the future.

Chapter 10

Bibliography

- Aber, J. S., Marzloff, I., Ries, J. B., and Aber, S. E. (2019). *Small-Format Aerial Photography and UAS Imagery*. Elsevier.
- Advanced Photonix Inc. (2014). *TO-18 Hermetic Photocell*.
- ArcGIS API for Python (n.d.). How single-shot detector (SSD) works? <https://developers.arcgis.com/python/guide/how-ssd-works/>. Accessed on: April 27, 2020.
- Arduino (n.d.). ARDUINO MKR1000 WIFI. <https://store.arduino.cc/arduino-mkr1000-wifi>. Accessed on: April 24, 2020.
- Auburn (n.d.). Heat Transfer. https://www.auburn.edu/academic/forestry_wildlife/fire/heat_transfer.html. Accessed on: February 28, 2020.
- Deakin Research (2018). Flaim systems introduction. <https://www.youtube.com/watch?v=qetFVU9G4iw>. Accessed on: April 24, 2020.
- Dede, C. (2009). Immersive interfaces for engagement and learning. *Science*, 323 5910:66–9.
- Dupire, B. and Fernández, E. B. (2001). The command dispatcher pattern.
- FLAIM Systems (2019). Flaim trainer overview. <https://www.flaimsystems.com/flaim-trainer/>. Accessed on: April 24, 2020.
- Flynt, J. (2019). What to do When your 3D Printer Nozzle Keeps Clogging. <https://3dinsider.com/3d-printer-nozzle-clogging/>. Accessed on: April 24, 2020.
- Goldschmidt, B. (2019). 3D Printer Bed Adhesion: All You Need To Know. <https://all3dp.com/2/3d-printer-bed-adhesion-all-you-need-to-know/>. Accessed on: April 24, 2020.
- Hannah (2019). Fires - How they start and how they spread. <https://firearrest.com/fires-how-they-start-and-how-they-spread/>. Accessed on: May 17, 2020.
- Hewlett-Packard (n.d.). HP Z VR Backpack G1 Workstation Specifications. <https://support.hp.com/in-en/document/c05757035>. Accessed on: April 26, 2020.

- Hullette, T. (2019). 3D Printer Over-Extrusion: 3 Simple Solutions. <https://all3dp.com/2/over-extrusion-3d-printing-tips-and-tricks-to-solve-it/>. Accessed on: April 24, 2020.
- Jacob, R., Girouard, A., Hirshfield, L., Horn, M., Shaer, O., Solovey, E., and Zigelbaum, J. (2008). Reality-based interaction: A framework for post-WIMP interfaces. In *Proc. CHI 2008*.
- Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer.
- Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. C. (2016). SSD: Single Shot Multibox Detector. In *European conference on computer vision*, pages 21–37. Springer.
- Marsden Fire Safety (n.d.). Fire extinguishers. <https://www.marsden-fire-safety.co.uk/resources/fire-extinguishers>. Accessed on: February 20, 2020.
- MathWorks (n.d.). What is Object Detection? 3 things you need to know. <https://nl.mathworks.com/discovery/object-detection.html>. Accessed on: April 27, 2020.
- Merriam-Webster (n.d.a). Stereo camera. <https://www.merriam-webster.com/dictionary/stereo%20camera>. Accessed on: April 25, 2020.
- Merriam-Webster (n.d.b). Virtual reality. <https://www.merriam-webster.com/dictionary/virtualreality>. Accessed on: April 25, 2020.
- Microsoft (2018). What is mixed reality? <https://docs.microsoft.com/en-us/windows/mixed-reality/mixed-reality>. Accessed on: April 25, 2020.
- Niehorster, D. C., Li, L., and Lappe, M. (2017). The accuracy and precision of position and orientation tracking in the htc vive virtual reality system for scientific research. *i-Perception*, 8(3):2041669517708205.
- One Bonsai (n.d.). How VR Achieves Presence. <https://onebonsai.com/blog/how-vr-achieves-presence/>. Accessed on: May 3, 2020.
- Qualcomm (n.d.). XR is the future of mobile computing. We are making mobile XR a reality. <https://www.qualcomm.com/invention/extended-reality>. Accessed on: April 30, 2020.
- Rubin, P. and Grey, J. (2020). The WIRED Guide to Virtual Reality. <https://www.wired.com/story/wired-guide-to-virtual-reality/>. Accessed on: April 18, 2020.
- Schlager, B. (2017). Building a Virtual Reality Fire Training with Unity and HTC Vive. In *CESCG 2017: The 21st Central European Seminar on Computer Graphics*.
- Sharan, L., Rosenholtz, R., and Adelson, E. (2010). Material perception: What can you see in a brief glance? *Journal of Vision - J VISION*, 9:784–784.

- Silicon Laboratories Inc. (2019). *Si721x Field Output Hall Effect Magnetic*.
- Stereolabs Inc. (2020). ZEDManager.cs. github.com/stereolabs/zed-unity/blob/master/ZEDCamera/Assets/ZED/SDK/Helpers/Scripts/ZEDManager.cs. Accessed on: April 29, 2020.
- Stereolabs Inc. (n.d.a). Depth sensing overview. <https://www.stereolabs.com/docs/depth-sensing/>. Accessed on: January 21, 2020.
- Stereolabs Inc. (n.d.b). How to use TensorFlow with ZED. <https://www.stereolabs.com/docs/tensorflow/>. Accessed on: May 1, 2020.
- Stereolabs Inc. (n.d.c). ZED Mini - Mixed-Reality Camera. <https://www.stereolabs.com/zed-mini/>. Accessed on: January 21, 2020.
- TensorFlow (2018). Configuring the Object Detection Training Pipeline. https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/configuring_jobs.md. Accessed on: May 1, 2020.
- TensorFlow (2020a). Tensorflow Object Detection API. https://github.com/tensorflow/models/tree/master/research/object_detection. Accessed on: April 27, 2020.
- TensorFlow (2020b). TFRecord and tf.Example. https://www.tensorflow.org/tutorials/load_data/tfrecord. Accessed on: May 1, 2020.
- Texas Instruments (2016). *DRV5033 Digital-Omnipolar-Switch Hall Effect Sensor*.
- Unity Technologies (n.d.). Game engines-how do they work? <https://unity3d.com/what-is-a-game-engine>. Accessed on: April 28, 2020.
- Valve Corporation (2017). SteamVR Tracking Technology Update. <https://steamcommunity.com/games/steamvrtracking/announcements/detail/1264796421606498053>. Accessed on: May 1, 2020.
- Valve Corporation (2019). *SteamVR Unity Plugin*.
- Valve Corporation (n.d.). SteamVR Tracking. <https://partner.steamgames.com/vrlicensing>. Accessed on: May 1, 2020.
- VIVE (n.d.a). VIVE Enterprise. https://enterprise.vive.com/eu/?_ga=2.259048781.650489788.1589580668-36822316.1589580668. Accessed on: May 15, 2020.
- VIVE (n.d.b). VIVE Enterprise. <https://enterprise.vive.com/eu/product/vive-pro/>. Accessed on: January 21, 2020.
- VIVE (n.d.c). VIVE Wireless Adapter. <https://www.vive.com/eu/wireless-adapter/>. Accessed on: April 26, 2020.
- VR Awards (2018). 2018 Winners and Finalists. <https://vrawards.aixr.org/winners-and-finalists-2018/>. Accessed on: January 21, 2020.

Appendix A

Calculations of the flow rate selector ranges

For the flow rate selector, two analog hall sensors are used. Because the output of these analog sensors fluctuates quite a bit, some ranges needed to be determined. There are eight different settings on the flow rate selector: 115, 180, 230 and five gradients in flush. An output value is determined for every setting. For each setting, the output value of both sensors is read. This is repeated five times. The values are shown in table A.1. For each test, the average of both sensors is calculated. These values are shown in table A.2. Next, the overall average value is calculated for each setting.

Table A.1 Measurements taken from the analog hall effect sensors (A1 and A2)

	Test 1		Test 2		Test 3		Test 4		Test 5	
Flow rate	A1	A2	A1	A2	A1	A2	A1	A2	A1	A2
115	354	310	372	314	372	313	362	313	355	310
180	365	327	392	354	381	329	372	329	372	336
230	382	350	401	374	395	352	387	350	386	356
Flush1	419	396	430	417	430	403	426	403	422	403
Flush2	447	440	453	450	456	442	454	444	449	444
Flush3	436	465	472	471	469	463	467	465	463	466
Flush4	479	483	483	482	482	481	480	483	482	485
Flush5	487	486	488	487	487	486	485	489	487	492

Table A.2 also contains the mean absolute deviation (MAD), calculated with formula A.1, where $m(X)$ is the overall average and x_i are the averages of each test. Using this average deviation, the ranges are calculated. The lower bound is the average deviation subtracted from the average. The upper bound is the average deviation added to the average.

$$MAD = \frac{1}{n} \sum_{i=1}^n |x_i - m(X)| \quad (\text{A.1})$$

Table A.2 Average of both sensors for each position of the flow rate selector

	115	180	230	Flush1	Flush2	Flush3	Flush4	Flush5
Test1	332	346	366	407.5	443.5	450.5	481	486.5
Test2	343	373	387.5	435	451.5	471.5	482.5	487.5
Test3	342.5	355	373.5	416.5	449	466	481.5	487
Test4	337.5	350.5	368.5	414.5	449	466	481.5	487
Test5	332.5	354	371	414.5	402.5	464.5	483.5	489.5
Average	337.5	355.7	373.3	417.2	439.1	463.7	482	487.5
Avg. deviation	4.2	6.92	5.76	7.12	14.64	5.28	0.8	0.8
Lower bound	333	349	368	410	424	458	481	487
Upper bound	342	363	379	424	454	469	483	488

The intervals do not follow each other up nicely, so when there is a gap or the bound overlap, the average value of the two bounds is used. The final result is the following:

- 115: 333 - 345
- 180: 345 - 365
- 230: 365 - 395
- Flush 1: 395 - 424
- Flush 2: 424 - 456
- Flush 3: 456 - 475
- Flush 4: 475 - 485
- Flush 5: 485 - 488

FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
CAMPUS GROEPT LEUVEN
Andreas Vesaliusstraat 13
3000 LEUVEN, België
tel. + 32 16 30 10 30
iiw.groept@kuleuven.be
www.iw.kuleuven.be

