DIONISO - A proposal for leveraging HoloLens2 in simplified client-server based collaborative mapping for Search and Rescue Applications

Francesco Ganci

Department of Computer Science, Bioengineering, Robotics and System Engineering (DIBRIS)

University of Genova

Supervisor

Carmine Recchiuto, Antonio Sgorbissa

Master Degree in Robotics Engineering

December 19, 2023

Abstract

Starting from a former proof of concept, main purpose of this work is to introduce a wider viewpoint on the applicability of *Mixed Reality* in the large draw of Search&Rescue operations.

As first step, this work points the light on practical aspects in order to determine a realistically applicable solution in terms of needs of the rescuers and operational challenges. First respondents have been taken of primary consideration. Environmental aspects have been evaluated as well, e.g. the almost complete lack of connectivity, in contrast with a solution relying too much to the cloud to share informations. Also organisational aspects are evaluated here, for understanding how a rescue team is made and how organisations are managed.

All these aspects have been condensed and transformed in project requirements, involving both a client side realized in HoloLens2 and a server side with a relational database and a simple API implementation.

About the client side, the main problem considered is the mapping of the environment with a essential approach. A localisation algorithm has been implemented, running on HoloLens2 app, able to store graph-based mapping based on the internal odometry. Networking capabilities have been added at later time, for sharing informations with a server side taking advantage of any available spot of connectivity. User's activity have been represented with a simplified data model easy to share ant to reimplement in devices different than HoloLens2.

As support from the server side, a relational data model has been implemented along with a novel protocol (the D.U. protocol) able to fuse together mapping data from different devices, and able to keep the amount of exchanged data as less as possible with a careful control of what is known by the other devices in that moment.

To end the path, a rich framework for performances assessment have been developed, then employed for testing the final app. The design approach hereby applied will be more general, not only related to HoloLens2, but also *portable* to other devices, creating a common way of operating, and beside this, interoperability.

Keywords: Search And Rescue, HoloLens2, Collaborative Mapping, Three-tier Cloud-based architecture, Wearable Robotics, Data Fusion

Contents

1 Motivations and First Steps **1**
- 1.1 Introduction . . . 1
- 1.2 Related Works . . . 2
  - 1.2.1 About the Role of First Respondents . . . 4
  - 1.2.2 Need for a common Knowledge Base for SaR . . . 5
  - 1.2.3 SaR and Connectivity . . . 6
  - 1.2.4 Main paper selection . . . 8
- 1.3 Development Technology and Design . . . 8
  - 1.3.1 Development process and Metodology . . . 11
- 1.4 Structure of this Report . . . 12

2 Visualisation and Mapping **13**
- 2.1 Problem Statement and Motivations . . . 13
  - 2.1.1 HoloLens2 Positioning System . . . 13
- 2.2 Model: Accessibility Graph . . . 14
  - 2.2.1 The Waypoint Data Structure . . . 15
  - 2.2.2 The Path Data Structure . . . 16
  - 2.2.3 The Positions Database . . . 16
  - 2.2.4 The Localisation Problem . . . 16
  - 2.2.5 Mapping and Sampling Methods . . . 17
- 2.3 Visuals . . . 18
  - 2.3.1 Visualisation Modes and Interactions . . . 18
  - 2.3.2 Visuals Pattern Overview . . . 19
  - 2.3.3 Simple Path Visual . . . 20
  - 2.3.4 Surroundings Visual . . . 21
  - 2.3.5 Portable Minimap Visual . . . 22

3 Localisation and Mapping Algorithms for HoloLens2 **27**
- 3.1 Problem statement and Motivations . . . 27
  - 3.1.1 Localisation System Requirements . . . 28
- 3.2 Direct GeoLocation System . . . 29
  - 3.2.1 Formulas for Geolocation . . . 29
    - 3.2.1.1 Polar coordinates to Cartesian Absolute Coordinates . 30

3.2.1.2 Cross Product in left-handed reference systems 31
3.2.1.3 Transformations for left-handed reference systems . . . 31
3.2.1.4 Absolute to relative coordinates 32
3.2.2 Geolocalisation Solution Structure 33
3.2.2.1 Calibration Algorithm 34
3.2.3 Comments on Practical tests 34
3.2.3.1 Shifting point of view 35
3.3 Odometry-based Localisation System 36
3.3.1 Solution Structure 36
3.3.2 The Positions Database 37
3.3.2.1 Localisation and Tracking Solution Design 38
3.3.2.2 Basic Version of Dynamic sort 38
3.3.2.3 Cluster Optimisation 39
3.3.2.4 Max Index Optimisation 40
3.3.2.5 Creation of new positions 41
3.3.3 Calibration Procedure Development 43
3.3.3.1 Calibration Implementation 43

4 Cloud Integration **47**
4.1 Problem Statement and Motivations 47
4.1.1 Connectivity in SAR applications 48
4.1.2 Security in SAR applications 48
4.1.3 Collaborative mapping and Efficiency 49
4.2 The JSON Import-Export 50
4.2.1 Solution Structure 50
4.2.2 Export Procedure 50
4.2.3 Import Procedure 51
4.2.4 Conclusions and results 53
4.3 Solution Structure Overview 54
4.4 Data Model Overview 55
4.4.1 Security Mechanisms Implementation 57
4.5 The D.U. Protocol - Server side 59
4.5.1 Protocol Design 59
4.5.2 Waypoint Staging Table 61
4.5.3 Support for Alignment Algorithm and Alignment Quality Heuristic 63
4.5.4 Paths Staging Table 64
4.5.5 Server Protocol Expected Usage 65
4.6 The D.U. Protocol - HoloLens2 Integration 65
4.6.1 Shared Keys and Stable Keys 66
4.6.2 High Level Protocol Orchestration 67
4.6.3 Download Implementation 69
4.6.4 Upload Implementation 71

5 Testing of the final Project and Results **75**
5.1 Main Key Performance Indicators 75
5.1.1 KPI Frame Rate 76
5.1.2 KPIs Hit Rate and Miss Rate 76
5.1.3 KPI Sort Quality 77
5.1.4 KPI Chasing Distance 79
5.1.5 KPI PosDB Load 80
5.1.6 KPI busy time 80
5.1.7 KPI Swaps per call 81
5.1.8 KPIs Request KB and Response KB 81
5.1.9 KPIs Latency and Minimum Bandwidth 82
5.2 Application Testing 82
5.2.1 Benchmarking, Reports and Logging 82
5.2.2 Simulated Experiments 83
5.2.2.1 Visuals Stability Testing 83
5.2.2.2 Positions Database Tuning 86
5.2.2.3 Straight Line Benchmark Tests 89
5.2.2.4 Waypoints Benchmark Test and Server Test 91
5.2.3 Device Experiments 93
5.2.3.1 Application Experimental Setup 93
5.2.3.2 Application Tests 94

6 Conclusions **99**
6.1 Results Recap 99
6.2 Device Limitations 101
6.3 Future Developments 102

7 Appendix 1 - Final Application Overview **107**
7.1 The Client Side 107
7.1.1 HoloLens2 App Structure and Guidelines 108
7.1.2 Available Main Packages 110
7.1.3 Signals and Events 116
7.1.4 Project-specific Scripts 116
7.1.5 Logging and Data Collectors 119
7.1.6 App Complete Arch and Startup 121
7.2 The Server Side 123
7.2.1 Development Tools and Architecture 123
7.2.2 Database Implementation Structure 124
7.2.3 API Container Structure 124
7.2.4 System Startup 126

8 Appendix 2 - Server Side Technical Details **127**
8.1 Naming Conventions 127
8.2 Fake Token Protocol 128
8.3 D.U. Protocol - Download Transaction 131
8.3.1 High Level Interface 131
8.3.2 Waypoints extraction 132
8.3.3 Paths extraction 133
8.3.4 Extraction filtering 133
8.3.5 Data collection and response 134
8.4 D.U. Protocol - Upload Transaction 135
8.4.1 High Level Interface 135
8.4.2 SQL Waypoints Processing 137
8.4.3 Paths pre filtering 141
8.4.4 SQL Paths Processing 142

References **148**

List of Figures

1.1 A image of Amatrice after the earthquake of year 2017 2
1.2 Azure Spatial Anchors feature, as represented by the official documentation 6
1.3 A first overall vision of the interesting parts of the organisation 9
1.4 The first implementation proposal 10
1.5 A common representation of the spiral model in SDLC 11

2.1 The current implementation of the visuals pattern, according to the model-view-controller design pattern. 20
2.2 How the simple path visual appears after a "long" exploration of the environment. 22
2.3 The logical structure of the minimap, shown as it appears during simulated run. 24
2.4 The minimap, as shown in a test on the device. Borders of the map are barely visible in this image; they are clearly distinguishable in live application. It shows all the known paths around the user: in this case, back, right and forward. The red zone represents the current user's position. 25

3.1 A representation of coordinates λ (longitude) and ϕ (latitude). 29
3.2 UWP Procedure to extract Geolocation 33
3.3 Representation of the distribution; the figure shows the reason why it has been chosen to take the double of the base distance: to have as less overlappings as possible between near zones. 42
3.4 During the calibration process, a audio signal is emitted to warn the user that the calibration is about to start, and a "jerkling" cursor is visualised, aligned with the user's line of sight; it should be used as a aim for controlling the orientation. The user points the cursor to a given target, and keeps the same position until the calibration snapshot has not been collected. 45

4.1 Anomaly found when importing data from a JSON; the algorithm gives some problems only while the user is walking during the import. 53

4.2 A diagram showing the current data model, simplified. Arrows go from the main version of the field in one table, to the related field in another one table. 56
4.3 Schema of Waypoints Staging Table 62
4.4 Maching Quality Heuristic, in the case of D=1.5m and b=2; it can be used to evaluate the quality of a matching of one uploaded point with another previously recorded one. 64
4.5 Schema of Paths Staging Table 65

5.1 Comparison between two ways to compute the KPI frame rate; in orange the first formula, and in blue the second formula. Values are obtained from a simulation 77
5.2 Typical trend of HIT/MISS rate, obtained in a situation where the system is working fine 78
5.6 Hit/Miss Vs. PosDB Load in straight line benchmark. 80
5.7 Test Map, used for development and testing in simulated environment. . 83
5.8 Visual Minimap during the visuals test, after explored the entire area . 84
5.9 Visual Minimap during the visuals test, after explored the entire area . 85
5.10 Estimation of device frame rate given the one from the simulation using minimap visual. 86
5.11 A possible anomaly, generate in test number 2 of the minimal tuning assessment. Due to the high velocity of the user, one waypoint have been generated "in the middle of" other two waypoints. 88
5.14 Server simulated test, third session: some anomalies are generated by the system during the download, due to the quick movements of the player. 92
5.15 The final map as result of different sessions mixed in one dataset. 92
5.16 The app Hands Menu. The leftmost column contains the three visuals. The right down button makes the leftmost info panel appear. 93
5.3 Trend of the KPI Sort Quality when the system is working properly. User is moving up and down across a straight line. Behaviour of the metric tends to be bounded after a while. 95
5.4 Trend of the KPI Sort Quality when the system is not working properly: in this case, the metric tends to diverge due to the continuous increasing of the number of points; the system is not able to properly follow the user's position. 95
5.5 The following diagram shows the comparison between the HIT/MISS trend and the Chasing Distance. The user is moving along a straight line in simualted environment. 96
5.12 Base distance 0.25m with no optimisations: the database is no more able to correctly follow the user's position, causing the generation of a enormous number of useless points. 96
5.13 Testing Environment for the Positions Database Tuning; it is pointed out the Unity component providing a live estimation of HIT/MISS ratio. 97

5.17 Path of the first test on the device, with user's perspective, values from internal odometry, and sampling from positions database. 98

7.1 Meaning of the symbols in the component diagrams in this chapter . . . 110
7.2 HL2 Package for Networking 111
7.3 HL2 Positions Database Package 112
7.4 HL2 Visual Packages 113
7.5 Sliding tool before the activation. In the image, the SceneUnderstanding map is "simulated" by a set of small random 3D shapes in the space of the minimap, i.e. under the same root. 117
7.6 Sliding tool after the activation. The sliding tool was a quasi-plane cube: the user can move it along the choosen dimension, in this case the Z axis. When enabled, all the other shapes are hid, except for the ones falling into the range of the tool. 117
7.7 Signals Pattern represented. The UnityEvent is a object containing all the references to the called object. Notice that the entire process happens in the same frame for each called component. 118
7.8 Package Schema: DiskStorageServices 120
7.9 Overall HoloLens2 System Architecture 122
7.10 Overall Server Architecture 123
7.11 API Transactions Architecture 125

LIST OF FIGURES

List of Algorithms

1 Simple Path Visual Logics 21
2 Surroundings Visual Logics 23
3 Positions Database Update procedure 37
4 Dynamic Sort Simple 39
5 Dynamic Sort Cluster 40
6 Dynamic Sort Cluster Max Idx 41
7 JSON Export Procedure 51
8 JSON Import Procedure 52
9 HoloLens2 D.U. Protocol Orchestration 68
10 Data Collection and PreFiltering on SarClient 70
11 PosDb Data Integration from Download 71
12 Data Upload on SarClient 72
13 PosDb Renamings Integration 73
14 Path Exclusion Procedure 134

Chapter 1

Motivations and First Steps

1.1 Introduction

The project, based on the works Recchiuto *et al.* [2017] and Testa [2022], is part of the DIONISO project, *"Tecnologie innovative di Domotica Sismica per la Sicurezza di Edifici e Impianti"* (Dioniso [2023]). Supported by MIUR (Italian Ministry of Education, Universities and Research), it has been proposed by the University of Messina as the next step of the PRISMA project, from 2018 to 2023. The project was divided into four macro topics: INSIDE, ESCAPE, LIFELINES, and RESCUE; this thesis work belongs to the RESCUE path, i.e. *"development of home automation and smart systems to identify people who have been buried by building collapses and are still alive"*.

Let's provide more context for this work. In emergency situations, first Responders may need to acquire in real-time a huge amount of geolocalized data about the surrounding environment, which can be partially or completely unknown. The base scenario is the earthquake of magnitude 6.2 that occurred in Italy at Amatrice in 2017 (image 1.1): almost 300 persons were killed in the earthquake, and aerial images taken by drones show houses and walls collapsed and swathes of the city completely flattened. The data to be acquired in this scenario may include, for instance, 2D/3D maps of the environment (e.g., required for damage assessment and for identifying dangerous areas needing a prompt intervention), the location of people needing assistance (e.g., injured or trapped in buildings), the availability of access routes for emergency vehicles, well as any other data that can help operators speeding up operations and increasing Search and Rescue efficacy. Analogous requirements may emerge in other natural or industrial disasters, or even in presence of crime and terrorist attack.

In this general scenario, this work will have the objective to develop and test algorithms for navigation, self localization and mapping using for instance wearable sensors embedded in the clothes of First Responders in the emergency area, or environmental sensors, working in conjunction with Augmented Reality. The maps built in real-time shall then be visualized on augmented reality eyewear (e.g., HoloLens) thus providing useful information for First Responders to explore the area the more efficient way as possible, and will be additionally usable by decision makers in a remote Control Room

Figure 1.1: A image of Amatrice after the earthquake of year 2017

for resource allocation.

As anticipated in the abstract, another PoC was already available at the beginning of the work, Testa [2022]; the practical first statement was to start from that project as base, and to find a way to integrate other data sources, e.g. wearable or environmental devices, and to improve the overall usability of the application. But, which is the best way to proceed? In order to be able to propose a reasonale solution, we first have to understand which are the main characteristics of a Search and Rescue operation.

1.2 Related Works

The first step of this project consisted in trying to deepen the aspect of SaR operations and the role of the first respondents: informations about the operative context, even if not too much detailed, are precious to make a proposal that can be realistically and effectively applied in rescue operations.

Research proceeded preferring papers and sources with a general point of view, especially linking the Augmented Reality with search and rescue, but paying attention to not go too much deep due to the very general research request, except for some cases, e.g. MANET networks. Literature reviews and surveys are preferred, in a range of years from 2019 to present. Some topics are not covered by this search criterion, hence other older articles have been included in order to have a moinimum of perspective. Technological references before 2019 are discarded, preferring more "modern" results.

First step consisted in listing all the keywords, taking into account our objective to extend the point of view. Here below a summary of the main keywords and papers considered:

- **HoloLens2** (Microsoft [2023f], Testa [2022], Palumbo [2022])
  - SAR HoloLens2 (Luksas *et al.* [2022])
- **Emergency Response**
  - Fire Emergency Response (Nunavath *et al.* [2016])
  - Earthquake Emergency Response (Recchiuto *et al.* [2017])
  - Urban Search and Rescue (Wang *et al.* [2018])
  - On-site Search and Rescue (Cao *et al.* [2021])
- **SaR operations**
  - Search and Rescue Operations (Nunavath *et al.* [2016], Maceda *et al.* [2018])
- **Disaster Assessment**
  - First Disaster Assessment (Maceda *et al.* [2018], Nunavath *et al.* [2016])
  - Disaster Risk Redution (Maceda *et al.* [2018])
  - Aided Decision Making (Huang *et al.* [2021], Cao *et al.* [2021], Maceda *et al.* [2018])
- **Delay Tolerant Network** (Grassi *et al.* [2023])
  - MANET, Mobile Ad-hoc Networks (Onwuka *et al.* [2012], Anjum *et al.* [2015], Agrawal *et al.* [2023], Ramphull *et al.* [2021])
  - Mobility Model (Aschenbruck *et al.* [2007])
  - MANET Security (Korir & Cheruiyot [2022])
  - WSN, Wireless Sensor Network (Kandris *et al.* [2020])
  - MANET performance evaluation (Reina *et al.* [2013], Moudni *et al.* [2016], Al-Shareeda & Manickam [2022])
  - VANET, Vehicular Ad-hoc Network (Onwuka *et al.* [2012], Anjum *et al.* [2015], Agrawal *et al.* [2023], Ramphull *et al.* [2021])
- **Disaster Robotics** (Williams *et al.* [2019])
  - Swarm Search and Rescue, Multi-robot search and rescue (Drew [2021])
- ***Other keywords considered***
  - Data Fusion, Sensor Data Fusion (Van Westen [2000], Cook *et al.* [2009])
  - Situation Awareness (Maceda *et al.* [2018])
  - Wearable Sensors (Seneviratne *et al.* [2017])
  - Adaptive User Interfaces (Lavie & Meyer [2010])

Nowthstanding this list doesn't provide a complete perspective, it is sufficient to build up a point of view. Noteworthy that a small number of papers precisely about the usage of HoloLens (and in particular HoloLens2) in a Search and Rescue context have been found, maybe due to the novelty of this kind of technology.

1.2.1 About the Role of First Respondents

To understand how we could improve the previous project, we need to understand first which are the situations that First Respondents have to face operatively, in order to find the most suitable features. So, could we determine, at least in general terms, which are the basic needs of a rescuer?

A good starting point can be found in Nunavath *et al.* [2016]; the main purpose of a Search and Rescue operation is *"to prevent loss of life and injury through search, locate and rescue persons in distress by alerting, responding, and aiding activities using public and private resources"*.

The article succeeds in providing a clear representation of the SaR situation, at least in firefight scenario; many aspects found here are more general in terms of approach to the SaR context.
Main aim of the first respondents is to assess the entity of the disaster in a timely fashon; time is a key factor, especially in saving lives. First respondents go around in the disaster area (not randomly, but following a well defined operative approact to the exploration), and try to take a picture of the environment, spotting first survivors (eventually trapped or injuried), victims, but also available paths and dangerous areas, in order to allow transit of vehicles, rescuers, and people involved in operations; every detail could result crucial for the coordination of rescue operations. Often, first respondents arrive to the disaster area before the rest of the organisation; the other ones join to the operations as soon as possible.

Vast Search and Rescue responses can involve a number of different organisations (Nunavath *et al.* [2016], Aschenbruck *et al.* [2007]): police, civil protection, paramedics, firefighters, and many others, all involved in rescue operations and coordinated by a operation center; the number of organisaètions clearly depends on the entity of the disaster. Moreover, organisations can also have at disposal autonomous agents such as drones, terrestrian robots, and others, for the simple fact that often it is more secure to send a robot in a dangerous area instead of jeopardizing a person's safety.

Fron this scenario, we could imagine right now some features that a rescuer could require: sending S.O.S. signals from the current position; sharing informations able to capture directly and quickly the situation, such as photos, 3D representations, and so on; notifying the current position, even in lack of GPS signal or networking in general; informing teammates about something important related to the operations (for instance, to indicate a way to follow to go in a hot zone); and more. To recall a concept proposed by Cao *et al.* [2021], this project refers mainly to *on-site Search and Rescue support*.

1.2.2 Need for a common Knowledge Base for SaR

Many papers emphatize the fundamental role of efficient information discovery and sharing for decision making in first disaster response (Maceda *et al.* [2018]), highlighting the strategical importance of being able to collect relevant information both about the environment (Testa [2022]) and from local witnesses (Grassi *et al.* [2023]), and to make them available to everyone in realtime.

Classical means for sharing informations are, for instance, walkie-talkie, but verbal interaction could be not precise enough, for many reasons and especially considering the pressure of the moment. To preserve the situation awareness and the attention of the rescuers, it is not feasible to collect "consciously" so many things; automation in collecting environmental informations and accessibility data, at least, becomes a key factor for improving the overall efficiency of the operations. In particular, accessibile zones and rescuers' positions in the disaster area are fundamental, referring to the problem of environment mapping, which is the main focus of this work.

From what said before, a common knowledge base is required to take track of the proceeding of the operations and to optimally manage operations; a common data space can provide a wide point of view on operations, as well as occasions for insights (preferrably automated) to be able to understand also aspects not clear at the first time, and detectable only after a collection of sufficient quantity of data from "agents" involved in the operations.

Maceda *et al.* [2018], even if not specifcally focused on first disaster assessment, shows how data can be prepared and then collected to support decision making in managing a emergency scenario applying SCRUM metodology. It underlies the need of a central source of data from where to extract as much disaster documents as possible in order to assess rapidly the possible entity of the disaster. This data source can be the merge of many particular integrated sources, each of them belonging to different government departments or private organisations.

Also Nunavath *et al.* [2016] puts the point on the importance to provide, right from start, a clear organisation of the informations, proposing in particular a relational data structure specifically created for that kind of emergency (in that case, fire scenario). Authors also discuss the main advantages of this specific approach, as well as challenges, especially in considering the topic of the data quality. In fact, integrating informations could be very difficult due to legal limitations, privacy limitations, witnesses not well informed or even under shock, and more.

As shown in Maceda *et al.* [2018], data can come from many sources, many of them specifically created by the government to face a possible emergency scenario, allowing rescuers to save time in finding out information about the context of the emergency. On the contrary, Cao *et al.* [2021] refers to a local collection of informations: this last one will be the point of view used for this project.

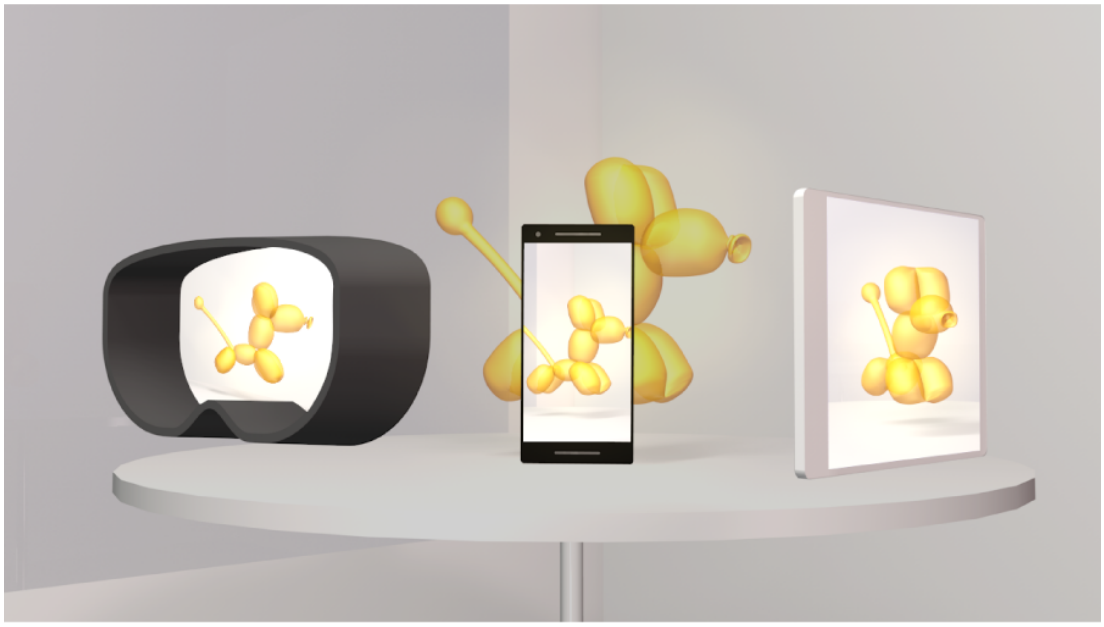
Interesting the work of Grassi *et al.* [2023] providing a centralisation of the informations able to integrate the data also from the users; part of the proposal is a data quality process able to distinguish fake/malitious alerts from the realistic ones. The problem of the reliability of the informations is deepened, considering wrong informa-

tions and intentionally wrong informations in particular, to overcomplicate a already tense scenario. Realtime availability of information is a fundamental point, found also in Recchiuto *et al.* [2017].

1.2.3 SaR and Connectivity

Regarding on server side, Testa [2022] proposes a solution entirely based on Azure, in particular Azure Spatial Anchors (Microsoft [2023c]) that is a service, specifically developed for HoloLens2, allowing multiple users to "see" the same object in Mixed/Augmented reality.

Figure 1.2: Azure Spatial Anchors feature, as represented by the official documentation



Azure offers a rich set of services allowing to create a more complex solution suitable also for supporting many other features like the one explored in Testa [2022]: storage (Microsoft [2023d]), databases/data transformation (Microsoft [2023b]), orchestration solutions (Microsoft [2023a]) and many other services.

Azure Spatial Anchors binds a virtual object (e.g. a set of meshes *located relatively to* another position) to one point in the space, making it available to other different device supporting Augmented Reality. It also provides consistency of positioning of the objects through different sessions (figure 1.2) so that the virtual object acts as a real one, staying in a position until someone does not move it. Interesting the example proposed in the Microsoft Documentation (Microsoft [2023c]) of the calendar in a meeting room: each users will see the same object in the same position even if from different devices. This is a really complicated objective to achieve, as this project shows in the following pages.

Despite it's a so powerful functionality, *it has been decided to not to use that* in this work, eventually considering to apply that kind of service in future iterations of the project. The main reason is the availability and stability of a connectivity in the disaster area, which is not sure at all.

When all the networking services are unreachable in disaster area (depending on the entity of the catastrophic event), SaR teams can provide a aid through a variety of solutions: to name just the most relevant according to the findings, Mobile Ad-hoc Networks (MANet), Vehicular Ad-Hoc Networks (Onwuka *et al.* [2012], Anjum *et al.* [2015], Agrawal *et al.* [2023]), and Wireless Sensor Networks (Kandris *et al.* [2020]); it happens often that different approaches are combined together. Especially Agrawal *et al.* [2023] provides a clear overview of the main networking solutions available in SaR context near to the state of the art.

MANet networks consist of a set of independent nodes, eventually in movement, communicating each other with a short range communication layer. According to Anjum *et al.* [2015], MANet networks have many characteristics suitable for SaR:

- Infrastructure-less
- Self-configurable and easy to deploy and use
- Dynamic, self-adaptive: when a node losts another one, it tries to find the nearest one capable of carrying the message
- Scalable: the network can adapt itself to more nodes or less nodes

These points make MANet a good solution in lack of connectivity, allowing rescuers to communicate and share informations even when the disaster is such as to render the network inoperative throughout the area.

The dynamic behaviuour of this solution is the main pro, but also the main cons, since the need for continuous adaptation causes lacks in connectivity and packet losses, slowing down the connection, so the performances won't be the same of a classical network infrastructure. Aschenbruck *et al.* [2007] and Reina *et al.* [2013] give the idea about one of the main problems of this network, that is the fact that the nodes move around; performances are related to the way the nodes move in the space, so mobility models are provided by many researchers, along with simulators, in order to estimate the performances of the networks implementations given the coordination of the operations. Routing in particular represents the most difficult topic for making a MaNet working fine.

In conclusion, networking is something unstable in Search and Rescue applications, and crucial at the same time since it enables all the components of the team to share potentially precious data. The present solution will cope with this problems acting on two ways:

- Minimizing the number of informations shared between the systems, i.e. optimizing the network usage;

- Making the device able to operate mostly offline when no connection is detected, as much as possible.

Since it is difficult to evaluate the amount of (useful) information exchanged by HoloLens2 using a service such as Spatial Anchors, it has been decided to not to use it, for searching a different model able to better cope with the intrinsic network unstability. To not to use Azure has also the advantage to allow "on-premise" solutions, especially when a VaNet network is applied: for instance, this choice allows to install a database in a vehicle connected to the network as a stand-alone application supporting on-site operations.

1.2.4 Main paper selection

In conclusion, to give a basic shape of the final application, mainly three articles have been considered as "source of inspiration":

- Luksas *et al.* [2022], proposing some interesting characteristics that the final application should have, as well as suggesting a research and specification modelling process related to the real practise of rescuers.
- Nunavath *et al.* [2016]; it has been chosen not precisely for the specificities of the results, but for the useful informations about the environment of SaR and for the proposal to have a main server integrating all the informations about the emergency. It is a good research starting point as well, for the papers contained in its stte of its state of the art.
- Testa [2022], which is the starting point of this work.

About the HoloLens2 part, this proposal will be mainly based on Luksas *et al.* [2022], the only one article presenting a implementation in HoloLens2 of a system supporting search and rescue, even if lacking of specific details about the real final implementation.

Starting from interviews with rescuers really employed in search and rescue operations, Luksas *et al.* [2022] detects a set of features that can be implemented in Mixed Reality. Scene Understanding can be seen as part of the application; so, the direction is to have a system integrating this kind of functionalities in a more extended framework able to combine different kind of features.

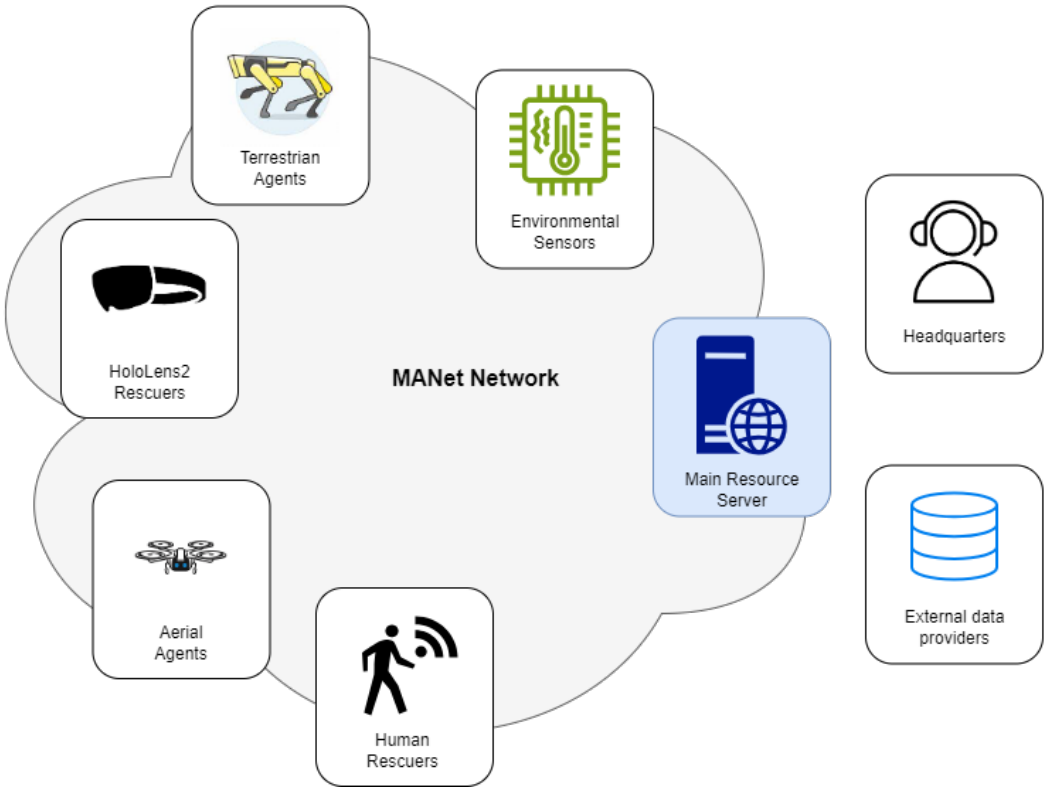
1.3 Development Technology and Design

From the previous discussion, let's put together all the informations to obtain a former idea of the structure and functionalities of the application.

One of the main points is that the same solution has to consider different agents, which have to cooperate together to achieve the same objective. A central knowledge base have to be built to be used both by agents and by a control unit able to provide a plan of the operations based on previously available data in combination with the data

from the first respondents. Information may be available in near-realtime, i.e. as soon as possible after its creation.
Figure 1.3 gives a idea of the situation: there's a central computing service, connected to all the other agents through a MANet network, and in communication with the headquarters as well as integrating data from external data providers as well. This is a very general representation: the network implementation can be more complex, as discussed before.

Figure 1.3: A first overall vision of the interesting parts of the organisation

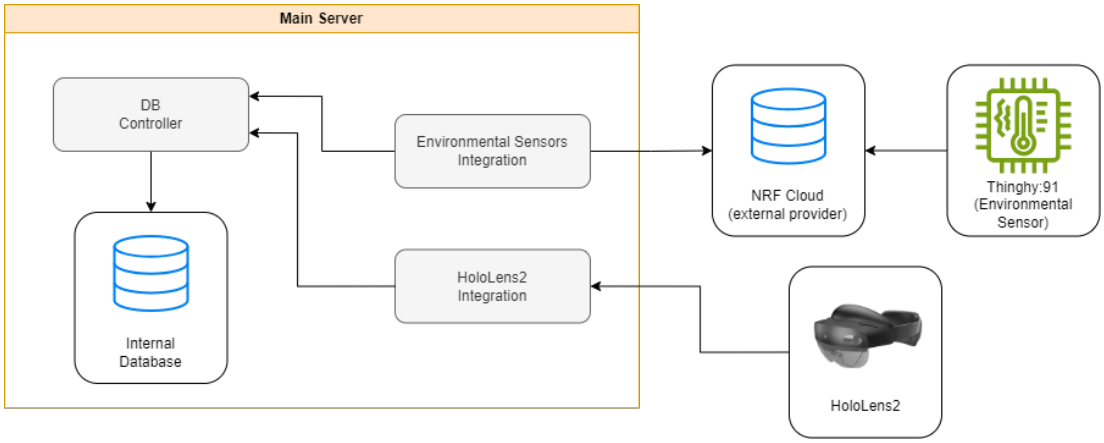


Data can be exchanged by many sources: from environmental sensors, to robots/autonomous agents, to other Mixed Reality devices, and to wearable devices carried by common rescuers. Data can also come from external sources, to provide as many useful informations as possible in short time. The control centre interacts with the central server, driving all the members of the organisation by processing informations and providing updates, even from other non-technological sources such as informations by the survivors. Of course, the server itself should also provide automated data processors to extract insight from data in order to not overload the control centre and to provide hints in the clearest form.

Figure 1.4 represents the first idea of the application, limited to a single HoloLens2 device, integrated with a environmental sensor through a server endowed with simple

data integration modules and a internal database. The main objective was to collect sensors data from Thinghy:91 (environmental sensor) and to visualise them into HoloLens2 with a suitable integration.

Figure 1.4: The first implementation proposal



The flow represented in 1.4 derives from a way of reasoning which will be very important in the following sections. Let's imagine to have to integrate Thingy:91, that is a environmental sensor able to measure things like temperature, air pressure and quality, humidity, and so on (Semiconductor [2023]); by construction, this sensor has its own connection, and uploads the data inside a third-party cloud server (NRF Cloud): data can be extracted by this database through a API call. The objective is to "see" the values of the sensor in augmented reality, for instance as a big popup on the device. Since it is possible to obtain values only from that server, we could come up with a HoloLens2 application able to periodically ask the server through API for new informations from the device, but it is a poor solution for a number of reason: to cite some of them, it will complicate and overload the HoloLens2 application, limiting performances; and, more important, we could be more interesting to share information to all the devices instead of creating a dicrect channel for HoloLens2. To propagate informations, we could build a server, in charge of collecting periodically informations and providing a API enabling all the devices in the network to easily get data from the sensor without worrying about the details of the service, and maybe having also other informations associated, depending on the data model used for integration. This is the starting point.

This simple proposal can be extended including other components such as data processors (backend applications running in background and processing informations in near-realtime in order to extract any helpful insight), integration APIs with other devices such as drones, and dedicated services for importing or exporting data, to cite some of all the possible variants. Important to notice that data processors can perform image processing for instance, or checking the activity status of a device to detect for example a endangered operator which didn't manage to provide a SOS; also for the

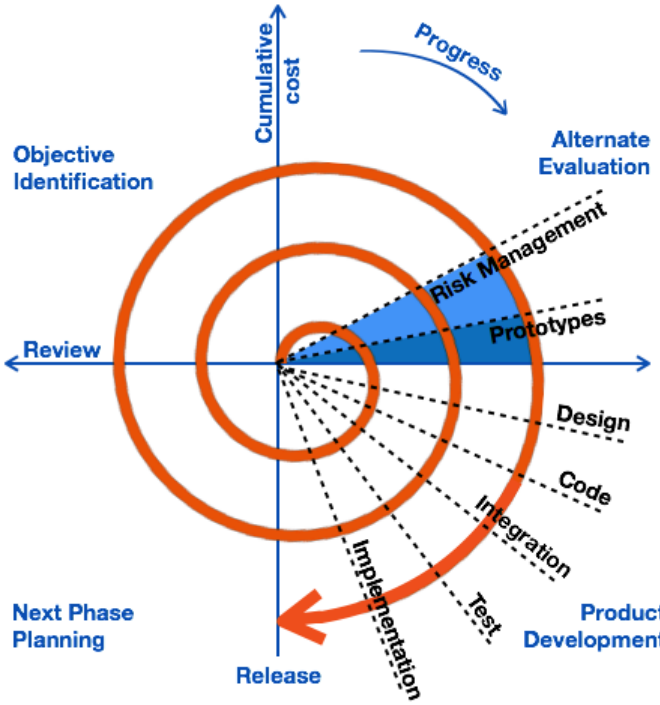
problem of mapping, information provided in raw format can be processed to get a clean map of the surroundings.

Talking about the client side, the only considered system will be HoloLens2. As requisite, the device should be able to work offline, taking advantage of any connectivity spot and exchanging as less data as possible (less, but effective). The main functionality explored here is the mapping, which has been implemented leveraging the user's activity in a completely automated way in order to enable the device to be "aware" of the position during the exploration of the disaster environment.

1.3.1 Development process and Metodology

As overall development strategy, the spiral model has been applied (Alshamrani & Bahattab [2015]), divinding the steps of the project in iterations with objectives of growing complexity, iteration by iteration (figure 1.5).

Figure 1.5: A common representation of the spiral model in SDLC



First iterations have been focused on the HoloLens2 side, from the development of the main development guidelines, to a first version of the AR application with a working tracking/mapping system, as well as guidelines for implementing and integrating visuals.

Subsequent iterations focused on the server side. Also in this case, the work started from simple things and in particular from the making of a modular architecture allowing

to make easy the management of basic processes such as logging and database requests. Next step covered the implementation of a collaborative mapping system running on the server. Client server intgration has been developed next.

The last step for this work is the development of a testing process aimed at assessing the performances and the stability of the final application.

1.4 Structure of this Report

The rest of this report will be structured as follows:

1. The *Chapter 2* (2) explains the basic **data model** employed for tracking the user activity, providing some basic informations about the implementation at high level, to pass rapidly to the **user interface** proposed under the scope of this work.

2. The *Chapter 3* (3) provides details about the **localisation and mapping system** that has been designed and then developed in HoloLens2, starting from the results about **geolocation based localisation** towards a localisation system purely based on **internal odometry**, from the mathematical settlment to the deveolopment of the system plus the **calibration mechanism**.

3. The *Chapter 4* (4) deepen the **server implementation** and the **collaborative mapping algorithm** developed for the project. It will be discussed the main ideas behind the implementation of the **D.U. Protocol**, from the design of the server implementation to the integration with HoloLens2. More details can be found in appendix 2.

4. The last *Chapter 5* (5) provides a **overview of the final project** from the guidelines to the final implementation as a virtual lab. After that, a framework for **performances evaluation** is proposed, starting the analysis from the most critical parts of the application towards a set of **benchmark tests** for assessing performances and stability of the system.

5. The *Concusions section* (6) will sum up the results of the work, as well as the feasible improvements and next steps of the project.

In the appendixes, the reader can find more technical informations about the project:

- The *Appendix 1* (7) provides a more technical point of view about how client and server are structured, also providing diagrams to represent the architecture.

- The *Appendix 2* (8) give more details about the most relevant server-side implementations, such as the two transactions of the D.U. Protocol.

Chapter 2

Visualisation and Mapping

2.1 Problem Statement and Motivations

The two main matters are: to *model the user's activity*, and to *visualise it*. The data model has to be as efficient and easy-to-build as possible. It should be possible, in particular, to integrate and extract data in the most light way, with a *on-the-fly processing*.
Scene Observer/Understanding is not well suitable for such set of requirements, since the model provided is too much detailed, and less controllable.

Another requirement is that the model has to be easy to exchange with other devices. It is of primary importance to have the possibility to track in realtime where's the user, both for supporting the resucer with visuals, as well as to make easier some future functionalities on server side. This is another reason to prefer a simplified model instead of SceneUnderstanding.

The approach is to start from the simplified data, aiming at providing a structure to be enriched later with more details. The rescuer goes around, and, in the meanwhile, it notices some elements which can be recorded in a structure attached to its current positions. In a nutshell, we're trying here to find a skeleton for the data, and then to collect more details to attach to the points of the simplified structure when possible. Not all the details are really necessary indeed, and ofter there's no time enough nor computational power to notice all the details. Collecting less informations has also the advantage to make the data more clean and then simple to interpret and share, even for other devices.

2.1.1 HoloLens2 Positioning System

The device has the capability to localise itself with a internal odometry referring to a relative position fixed when the device is turned on.
In terms of Unity implementation, it is easily accessible through the transform of the main camera (position plus orientation); the relative position (0, 0, 0) is obviously the origin of the frame that HoloLens2 created to obtain position, which has not fixed

coordinates with respect to a "global" frame. Orientation is provided in terms of quaternions. HoloLens2 implements a self-localisation based on *Parallel Tracking and Mapping* (PTAM, Jia *et al.* [2022]).

To map absolute coordinates, HoloLens2 does not have a GPS system or other similar mechanisms (Microsoft [2023f]): only WiFi localisation is possible, accessible through the Universal Windows Platform API. Unity does not provide natively a API for absolute localisation compaible with HoloLens2 (a case of "batteries not included").

Internal odometry of HoloLens2 is assumed to be always coherent in this work. More details can be found in the analysis of the results (chapter 5; see also device limitations in conclusions, 6).

2.2 Model: Accessibility Graph

Given the requirements, the graph data structure is the most obvious first choice: it provides a simple topological representation of the space, and it is simple to create and use for visualisation and tracking purposes.
In particular, the model hereby proposed is called *accessibility graph*. The idea is simple:

- Paths are divided into *waypoints*, linked by *paths*. Waypoints are positions collected as the user moves, using some sampling method.
- If the user moves from point A to another point B, it means that the target point B is accessible from the previous point A, hence *a new link is created* only if the point B is unknown. In a nutshell, only phisically traversed paths are stored, and only once. Graph is not oriented.

The main advantage is simplicity of its structure. It is possible to create a component that continuously checks the distance from the current position, integrating new ones when the distance increases over a threshold.

If finding whatever path from one position is enough for us, even if not optimal, this representation san work fine. But, more important to say that it enforces the rescuer to use well known paths during the exploration. To avoid risks, it is better to proceed along ways and paths which have been discovered and "tested before", since they are more secure with respect to some other alternatives.

The second rule should imply a acyclic data structure. For some activities, especially for recursive search, having a acyclic representation of the space is a vantage. Later this assumption has been relaxed: currently, implemented visuals can work well with cycles as well.

Another reason to prefer a representation with less links is the possibility to produce quickly more clean representations of the area. A structure with many links requires a iterative exploration with more operations than one with less paths. it is implied that we're discarding informations by preferring this way of proceed, but with a small gain in performances.

To summarize the properties of this data structure,

- A vertex represents an accessible position; a link represents the accessibiliy of one point fron another one;
- Accessibility is stated only when the point is discovered for the first time;
- Because of the concept of accessibility, the graph is not oriented;
- In its basic concept, graph would be acyclic, but we're not going to assume this fact;
- More data can be attached to the vertices and links of the graph; his allows us to deal only with the problem of mapping right now.

From now, waypoints will be the veritces of the graphs, and the links will be called paths. Let's discuss how the data structure is implemented.

2.2.1 The Waypoint Data Structure

Waypoint class implements a vertex of the graph. At simple level, this data structure provides these two elements:

- **A position** that is a vector of three components indicating a accessible position, recorded when the user discovered it at some time;
- **A list of near points** which will be a *list of paths* in this implementation.

Notice that if two waypoints are linked, they will contain a reference to the same path. The system currently tracks also other informations in addition:

- A *key*: a integer identifier for the position which must be unique (refer to the cloud integration, chapter 4 for further details);
- A *timestamp*, referring to the time the waypoint has been created for the first time;
- A *description* to excange useful informations between control unit and operator.

And more other data can be attached, even more complex. Almost all these informations are not integrated with visuals currently; attached data need to be managed better in other iterations of the project.

Each point has also a *radius*: an area is defined in terms of radius around one point, hence as a sphere around the sample; the user is at one waypoint if their position is inside that radius. For better measurements, the final system will take into account a cylindrical area instead of a spherical one, to not produce a new measurement if the user goes down with the head. One among position search and creation begins as the user goes out from that area.

2.2.2 The Path Data Structure

Path data structure represents a link between one waypoint to another one, meaning that it exists a phisically viable path from one point to another one.

A comment about the implemenation. To avoid data redundancy, paths do not record elements from the waypoints when they are created: for instance, distance can be asked to the path, but it is computed accessing to the positions of the two endpoints. This is a design principle: a *class should keep track of only the unique informations*, trying to avoid repetitions. This allows automatic propagation of informations, avoiding pointless updates.

2.2.3 The Positions Database

At least at C# level, informations can be stored just creating a overlay class containing the "root" of the data structure, i.e. the very first node. The program could search inside that each time time the user moves.
However, in most of the situations, we're not interested only in the current position and in the immediate near positions, but also to *all* the near ones, inclusing points that are in other parts of the graph, or not directly accessible with a simple or composite path.
Another reason to ask this lies in the fact that the data structure has some informations missing, because of the choice to keep track of the paths only when a new waypoint is discovered; hence, we have to accept that not all the near points are accessible.

To solve in paticular the last problem, we have to add another level of indexing/abstraction. It will exist a list containing all the waypoints defined during the exploration; this component, implementing both the list and the methods to access it in a optimal way, will be called the **Positions Database** and it will be a central, mission-critical component for localisation and mapping.

For the purpose of this chapter, it is enough to say that it exists a black box able to manage those queries. Further details about the implementation of this component can be found in the next chapter (3).

2.2.4 The Localisation Problem

In simple terms, the accessibility graph is a map of the environment; to support many visuals and tasks, it is necessary to implement also a system to localise the user inside this map. Localisation can be implemented as *efficient search of the nearest positions from a list*; in case the nearest position is too far, a new one has to be added.

How to make this search efficient? There are ways for optimisation. A first solution is to keep the list ordered with respect to the user's position, but, since the position changes frequently, sorting each time a long list of items is simply unacceptable from the point of view of computational resources. Let's consider a extensive exploration with thousands of points.

As solution, the list can be ordered little by little; a strategy could be to keep the list always semi-ordered depending on the distance between the current position and a point from the list. Doing so, the system tries to "follow" the user.
This is the main strategy implemented in this project, aiming at keeping things as simple as possible in terms of storage.

A throught about the performances. Keeping the structure *semi-ordered*, frame by frame, requires that the order does not change too much from one frame to another. How much could be correct this assumption?
We can assume that the rescuer typically has not to run during the exploration: it is legitimate to assume that the user walks the most of times. Choosing in addition a sustainable way to discover new positions during the exploration which is able to not generate too much points: this makes more reasonable such type of approach, since it is reasonable that the points around the user don't vary too much as the user goes on.

2.2.5 Mapping and Sampling Methods

Addressing the problem of mapping, to create a link between two positions, it is necessary to have something saying where the user is in that moment (first localisation, then mapping). Two different sampling approaches were evaluated in this work:

- A *time-based approach*: the system creates a new position with a fixed period, if needed.
- A *distance-based approach*: the system continuously evaluates the distance from the current position to the last localised one, trying to create a new waypoint as the distance becomes too much high.

The main advantage of the first approach is that, choosing a good sampling period, the system is able to track informations with acceptable performances, for the simple fact that there's no process running exactly on each frame. To insert a point when needed, it can be checked if the user is out of the previous position using distances.

Distance-based approach is more precise in terms of distribution. There are ways to make both the approaches to work fine: it is just a matter of tuning. But the time-based approach tends to produce anomalies in distances between two waypoints.
For instance, a user running instead of walking could produce too much long paths since the system is triggered with a fixed period; this kind of problem can be solved by a distance-based approach, which detects continuously the distance and can react in any time to the user's speed.

Another problem is represented by the frequent overlapping of near positions with time-based approach: for efficient tracking, the system is forced to create a new position the first time the user is found outside the previously identified position. A distance-based solution instead could be implemented in a way such that the distance is choosen so that waypoints are correctly distrbuted in the space.
In any case, having a mechanism to evaluate distances and track positions is required

in both the approaches. Choosing a time interval smaller and smaller makes the time-based approach to be equivalent to a distance-based one.

In the end, the final project implements a distance-based approach. The main drawback is related to performances: a process has to run in frame by frame, continouosly checking the distance. However, one advantage is to better cope with sudden rapid user's movements.

2.3 Visuals

At high level, to be able to implement a data visualisation with HoloLens2, we need a black box with these capabilities:

- Localisation capability
- Ways to access the current position, as well as graph traversing
- The system can issue events when the position changes

The problem to discuss now is about creating a pattern allowing to implement *visuals* on the data collected by the positions database. In particular, here are some problems to address:

- It is better to make some **abstractions** between the real form of the data and the way they are used, since otherwise the visualisation system will result in a too complex component (*spaghetti-code* or *lasagna-code* depending on your taste).
- Another useful abstraction to provide is a way to manage the storage of visualized objects **hiding the details of the instanced graphical assets**; we just want to instance something somewhere.
- One visualisation has to be implemented as a **independent module**, enclosing **all the logics in one place**, and making this component switchable.
- **Transition between different types of visualisations** have to be managed.

This project provides some very simple visualisations, but almost every visual that the human fantasy can create have to deal with the same types of problems.

2.3.1 Visualisation Modes and Interactions

HoloLens2 provides three main kind of interactions:

1. **Distance Interaction**, for instance pointing the finger somewhere, and "clicking" on it;

2. **Near Interaction**, for instance the possibility to put the hands on a virtual object and to change it; the possibility to touch virtual objects and to interact directly with them;

3. **Voice Interaction** refers to the possibility to give commands to the system using predefined keywords, and user's voice.

The first two interactions work with **hands gestures**: HoloLens2 systems are trained to detect particular shapes of the hands and movements, and to transform them in commands and interactions with the virtual world. **Voice commands** are the easiest way to develop for testing features, but in practice are very inefficient. Hence, a small menu has been implemented (the reader can find more details in the test section, chapter 5).

Three visuals have been proposed and fully integrated in the project, as listed below.

Simple Path Visual shows the waypoints and the links in Augmented Reality as the user walks around, drawing the path traversed until that moment. The user can see the zones already explored, and to follow them back if needed. Points remain visible until the user doesn't clean the visual.

Surroundings Visual shows the accessible paths around the user: a "branching" around the user is shown, making clear the known paths starting from the current position. The logic is a simple recursion within a fixed depth. Visual is refreshed as the user moves.

Portable Minimap Visual resembles a geographical map: a *virtutal map* appears in front of the user, placed in a way that allows the user to keep a good visual of the surrounding space. Movements of the figures of the map have been provided, in order to improve affordance.. It has been implemented as a variant of the Surroundings Visual.

Let's introduce them, one by one.

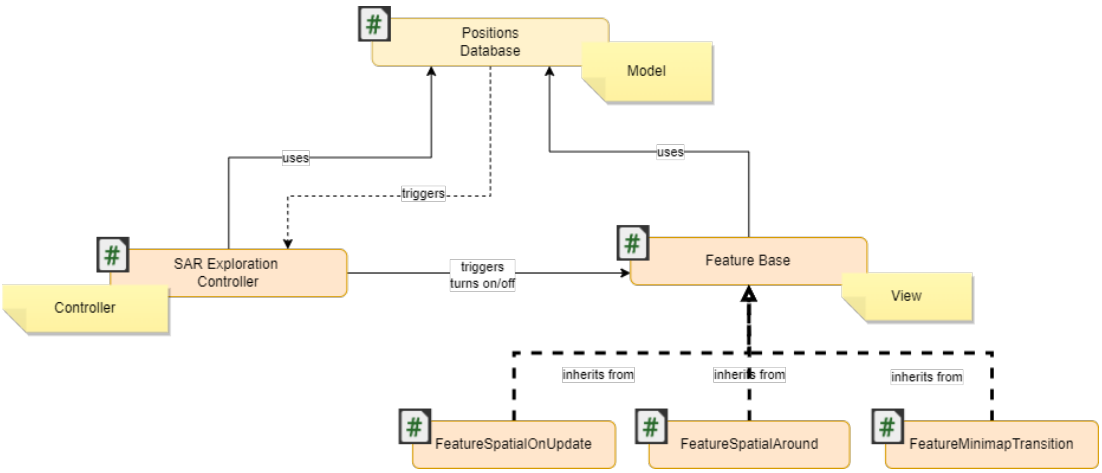
2.3.2 Visuals Pattern Overview

The implementation is based on the idea to separate different kind of problems in different blocks.
There are mainly three kind of functional entities here: one enclosing the logic of the feature (the so-called *Feature Base* class), another one controlling both the features and the transitions between different views (*SAR Exploration Controller* class), and a third block implementing the data source (the *Positions Database* component). Such a setting is a variant of the well-known *model-view-controller* design pattern.

Figure 2.1 represents how classes are combined in the project. Since all the visual features have almost the same basic characteristics, *Feature Base* is made to be inherited

by classes specialized with the implementation of the feature. For instance, each view can be enabled or disabled, or also runned when a event occurs.

Figure 2.1: The current implementation of the visuals pattern, according to the model-view-controller design pattern.



In the current implementation of the project, the controller class is just a interface able of taking into account the currently active visual and switching features through a bunch of callable EVENT methods.
As said in the conclusions (chapter 6), a more advanced implementation could consider state machines, able to react to different inputs as well as to better manage the semantic of the voice commands (managing context and persistent keywords for instance). In fact, since HoloLens2 just accepts some keywords, creating combination of commands requires a overlay aware of the meaning of the commands, and in particular able to distinguish between admissible and not admissible commands depending on the situation. The controller could consider also gestures as well, and evaluate them given a context. In conclusion, more development could be done here.

2.3.3 Simple Path Visual

Simple Path Visual provides a way to see the traveled path, drawn as the user walks. The logic is quite simple: a new point is spawned (using a drawer class) each time the database changes position only if it has not been spawned before; if needed, the new point is linked with the previously detected one.
Pseudocode 1 summarises the logics of this simple visual. Figure 2.2 gives a flavour of what the user sees.

The first phase visualizes the current position. The second one draws any link joining one already visualized position with the current one; in a sense, the algorithm proceeds "backwards", starting from the drawn point and trying to link the already visualised ones. Notice that the precise procedures used for drawing a virtual item are

Algorithm 1 Simple Path Visual Logics

Require: Drawer ▷ Utility for spawning markers
Require: Db ▷ Positions Database Reference
  currentZone ← Db.CurrentZone
  if point not yet spawned **then**
    Drawer.Spawn(*currentZone*)
  end if
  for all *pt* : path linked with *currentZone* **do**
    if *pt*.next already spawned but no link spawned **then**
      Drawer.SpawnPath(*pt*)
    end if
  end for

hidden in the class which have been generally called `Drawer`.

Integration at controller level is performed taking into account these simple transition rules:

- The functionality can be switched on and off anytime;
- Any marker is hidden when the user enables this feature as first one;
- Markers are not hidden changing visual from surroundings visual, keeping visible the branches around the first point where the visual is changed;
- A "close" command is provided, destroying all the spawned waypoints.

Noticeable that, in this implementation of the project, scene observer have been disabled, so that the user can see the path also through the walls; it could be useful to see the entire path at a certain point during the exploration. It is essentially a view for recalling the way to go back.

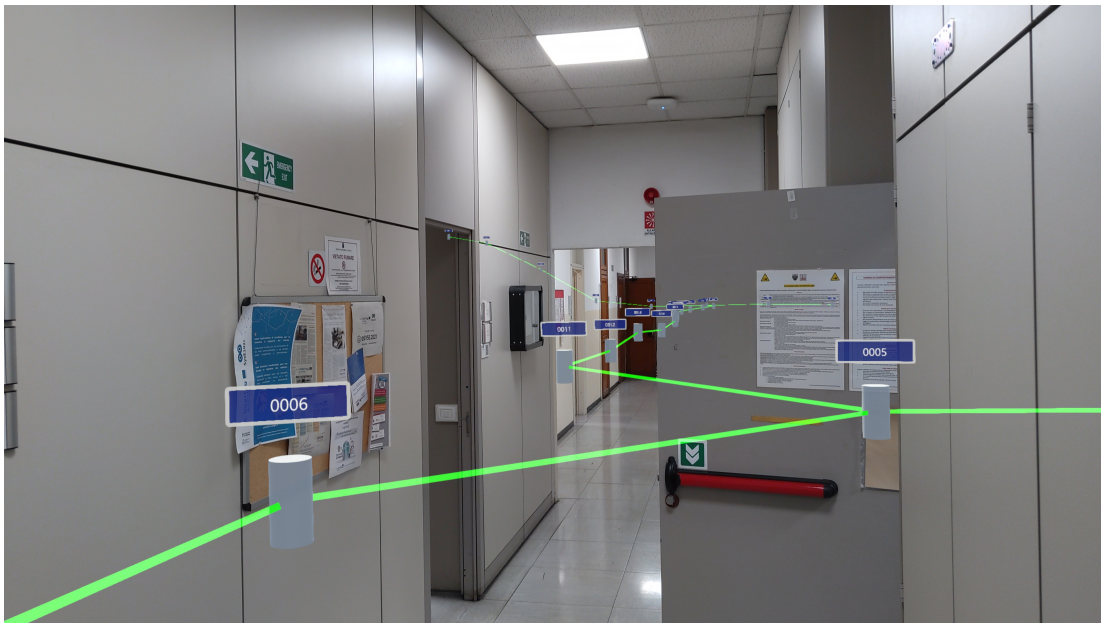
In future, this simple visual can be integrated with some other functionality making more clear the meaning of the points. For instance, the user could want to fix some checkpoint, or to mark some point in particular way. It requires to change the assets and to modify the drawer to handle different types of markers.

2.3.4 Surroundings Visual

Logic here is a bit more interesting than the one of "simple path" visual. This visualisation extracts data using a recursive search starting from the current position, and visualizing all the paths around the center with a given depth (parameter for the visual).

The pseudocode 2 summarizes the core logic of the visual. In the core function `DrawSurroundings()`, the parameter `userWp` is the user's position. To update the visual, the algorithm is applied starting from the current position of the user. The current

Figure 2.2: How the simple path visual appears after a "long" exploration of the environment.



implementation also visualises all the subgraphs near to the user as well, choosing all the near points as starting points and cycling over the set.

This implementation offers the possibility to manage depth of the visualisation with two voice commands: "less" (show less points) and "more" (show more points).

Transitions at controller level need to be addressed more deeply: in this implementation the system keeps the spawned points only switching visuals from surroundings to simple path.
Resources management can be improved as well. So far, the implementation just destroyes all the points before updating, and since the update of the visual is triggered only when the user changes zone, the changing frequency is not so high, making this solution acceptable. But it is not the best one: the system should consider also the already instanced points, hiding and showing instanced assets instead of creating and destroying them. This allows to preserve computational resources. As shown later, the system already supports this kind of resources management, which can be implemented with a small update on the drawer class (please refer to Appendx 1, 7).

2.3.5 Portable Minimap Visual

The objective is to create a small-sized visualisation of the surroundings; figure 2.4 shows this feature in action. Here are some characteristics developed so far:

- The map follows the user without restricting the user's field of vision; the experience resembles a *"paper map"*.

Algorithm 2 Surroundings Visual Logics

```
Require: Drawer                                  ▷ Utility for spawning markers
Require: Db                                      ▷ Positions Database Reference
  function DrawSurroundings(wp, userWp, dist, instances)
                              ▷ currently evaluated pos, user pos, remaining distance, set
    if instances is NULL then
      instances ← Set of strings
      userWp ← wp
      tag ← Drawer.Spawn(wp)
      instances.Add(tag)
    end if
    if dist > 0 or distance(wp, userWp) in drawable depth then
      for all pt : path in wp.links do
        nextWp ← pt.Next
        if next not yet spawned then
          tag ← Drawer.Spawn(nextWp)
          Drawer.SpawnPath(pt)
          instances.Add(tag)
          instances ← DrawSurroundings(
           nextWp, userWp, (dist - pt.Distance), instances
          )
        end if
      end for
    end if
    return instances
  end function
```

- The map is contained into a transparent box, to make clear the bounds of the object.

- Since the map is just a set of linked points, improving usability as far as possible is a priority. The map indicates the current position of the user in the map changing the color of the marker placed in the user's position.

- The map also can adapt its orientation according to the direction the user is looking at.

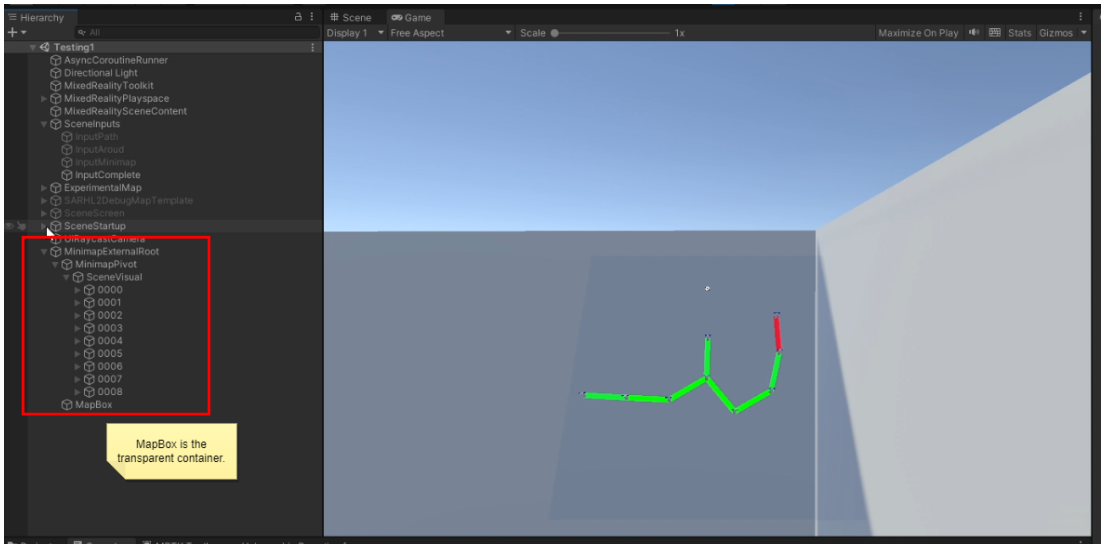
To implement the object, the basic idea is that a number of objects can be spawned under a common root enforcing a transformation to all the objects. In fact, as deepened in Appendix 1 (7), Unity models the world as a tree; each item is transformed with respect to its father object in terms of position, orientation and scaling. A first idea is to spawn all the markers and links of the map under a given object, setting a smaller scale.

First of all, the map has to follow the user, standing in fromt of them, slightly lower than the line of sight, and oblique. It is enough to look downwards to have a clean perspective of the map; the minimap should keep a intermediate position, to not to escape from the user's look, nor to limit the line of sight.

MRTK2 supports a particular type of scripts, called *solver* (Microsoft [2023e]). Solvers enable to implement movements of the virtual objects depending on position and orientation of the device. A custom solver has been implemented in this case, since it was not available among the default ones.
To implement the capability of pointing out the position of the user, another component has been applied: it listens for positions from the database and draws the current position in red.

The object is organized on three levels, as shows in figure 2.3: `MinimapExternalPivot`, `MinimapPivot`, and `SceneVisual`. The uppermost one (`MinimapExternalPivot`) is the root of the object, applying the overall scaling of the entire object as well as containing the solver.

Figure 2.3: The logical structure of the minimap, shown as it appears during simulated run.



The second level (`MinimapPivot`) comes into action as part of a strategy for windowing data in a small area. Let's consider a positions database covering an area of $1Km^2$: it is simply impossible to visualize all the points in this minimap with the constraint to keep a visibility to each point. The solution is to take less data; here comes the idea of re-using the surrounding visual: this logic already implements this data limitation.
But, it is not enough: points very far from the origin will be visualized very far also from the relative origin. So, keeping data "centered" is needed. The idea is to counter-

balance this behaviour applying a vector which is opposite to the ray vector from the relative origin to the relative position of the minimap content.
Now, the only problem to solve is: which point should be choose for applying such a transformation? If we have only one point, it is simple: let's take just that point and move it in the opposite direction with respect to its ray vector so that the point is centered. But, in the case of more than one point, a choice is to take the *average position* among all the points under the MinimapPivot. So the secondo root is moved from the origin to the opposite of the ray vector from the origin to the average position, so that the average position results to be located at the same point of the root of the entire object.

However, this is still not enough: the rotation is missing so far! Rotation can't be applied on the root of the object, otherwise the bounds of the map will rotate; and neither to the second layer. The second choice in particular produces a "clock movement" of the entire object, which is definitely not desirable.
So, a third level is placed, implementing the correct rotation axis; the right direction of rotation is opposite to the one of the user.

Figure 2.4: The minimap, as shown in a test on the device. Borders of the map are barely visible in this image; they are clearly distinguishable in live application. It shows all the known paths around the user: in this case, back, right and forward. The red zone represents the current user's position.

Transition problem is more challenging for this visual, since it requires to spawn and setup the entire object. In this implementation, at view level (see figure 2.1), the feature

has been divided into two different functional parts:

1. The first one, when enabled, builds the object in front of the user, trying to not destroy possible already instanced spawned points; it builds up the visual.
2. The second one implements the logic for updating the data inside the view, that is the well known surroundings visual; it performs the update and representation after the structure has been built.

By default, the component creating the map structure is made to instanciate both the static version of the map (not developed in this final project but very easy to create) as well as the dynamic version of the map (the one shows so far).

Chapter 3

Localisation and Mapping Algorithms for HoloLens2

Summary

The previous chapter (2) introduced the data model employed for tracking the user's activity in terms of positioning; on that line, this chapter shows how those requirements are implemented.
In order to enable HoloLens2 to interact with other devices in the same organization, it is needed to solve the problem of the localization of the device, mapping the local frame to a global one,making the device capable of sharing its position, as well as of understanding spatial informations from other devices.

All these problems are discussed in this chapter, making a bridge between the internal application logic and the server side.

3.1 Problem statement and Motivations

The main problem hereby considered is to find a approach to compose the accessibility graph, as well as to localise the user inside that structure.

We have to consider a more extensive point of view. To provide some terminology, we're going to use the expression *relative frame* to indicate the one created by the device when turned on. Talking about geolocation, we're going to use the expression *absolute frame* to denote a frame starting from the center of the Earth, assuming it spherical, with the vertical axis aligned with the line from south pole to north pole. And, as last term, talking about odometry-based localisation, a *operative frame* will be a reference frame fixed "on the SPOT", i.e. staying in one position and looking at some target; operative frame can be mapped in a absolute frame detecting its geo-location coordinates.

In terms of absolute coordinates, it is clear that there's not a direct connection between the two systems of coordinates: the HoloLens2 frame could generate a transform which is unknown when the device starts working, meaning that we have to come up

with a way to map the relative coordinates in a absolute reference system, otherwise building a *interoperability* between HoloLens2 and all the other devices and operators won't be feasible.

As use case which needs localisation capability, let's consider the integration with a simple environmental sensor, recalling a discussion presented in chapter 1. Let's consider in particular the case in which we simply want to visualise a window on top of the device, reporting the measurements. The sensor is also capable to localise itself in terms of absolute coordinates, so, with a API call, we can know the position of that sensor.
However, how could HoloLens2 correctly place this panel on the sensor if it does not know how to associate the two coordinates systems? And, to make thinks cumbersome, let's think that, at some time, this sensor could be moved somewhere else, so it is impossible to fix coordinates in advance: we need the updated position.
Another fundamental use case, even more significant, is the S.O.S. signal, feature which has not been implemented but that can be built on top of this work as well. To issue a S.O.S signal through HoloLens2, the device has to be able to send its *absolute position* to the other operators and autonomous agents, otherwise it is not possible to take actions quickly.
We can think on several other situations like the above mentioned twos. The previous work did not offer such a feature, so it has to be developed from scratch in this path.

3.1.1 Localisation System Requirements

The main aim is to build a localisation system able to exploit the capabilities of HoloLens2 to find a relationship (*transformation*) between relative and absolute frames.

There are other requirements to take into account. The first one is the connectivity: the system should be able to work offline as well. It is required that the application may be able to perform a static transform, starting from local coordinate, in the case the signal is down, preferring absolute localisation when a connectivity spot becomes available.
Networking in general is a critical aspect for Search and Rescue applications, since the signal can go down in any moment, so we need that HoloLens2 can be *as most autonomous as possible*.

Both the solutions discussed here use a initial *calibration step*. When the device is started, the transformation of coordinates is unknown; somewhat action by the user is required to fix the transformation, and after this action the system has a way to find its absolute position starting from its relative one, and viceversa.

A last requirement is that the device has to be able to manage a *data import/export*: this is required for the integration with other devices, as well as to integrate the device with a server.

3.2 Direct GeoLocation System

Despite the fact that it is not the final proposal of the system, the *IP-based localisation* has been the first "working" proposal made for this project. It has been developed and tested, obtaining unsatisfying results however. In spite of this, it is important to talk about this attempt, since many of the ideas and the concepts hereby explained have been re-applied later in the final proposal.

3.2.1 Formulas for Geolocation

Implementation of the Geolocation system requires the development of geometrical procedures; so, let's find formulas before going deep into the implementation details.

First step is notation. Let $< w >$ be the *absolute frame*, and $< u >$ the *Unity frame* (i.e. the relative frame), created anywhere when the device starts working.

The HoloLens2 Geolocation API provides a 2-components vector $\underline{gp} := (\phi, \lambda)$:

- ϕ is the *latitude*, which is the vertical coordinate; zero radiants indicate the equator.
- λ is the *longitude*, which is the horizontal coordinate; zero radiants points to the Greenwitch meridian.

For simplicity, let's assume these numbers in radiants, even if the UWP Geolocation module provides them in degrees, so a simple coordinates conversion will be needed. Let's assume, for the moment, that the data from the localisation API are *precise enough*. Figure 3.1 represents the frame with the coordinates.

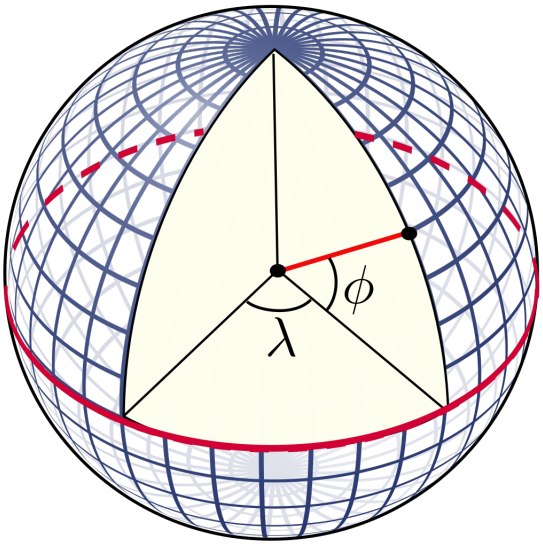


Figure 3.1: A representation of coordinates λ (longitude) and ϕ (latitude).

Noticeable that HoloLens2 does not support the height, so the $\underline{gp}$ vector will have only two components. As Earth radius, the *volumetric mean radius* have been assumed here (NASA [2023]). So, let $R = 6371.00$ Km be the Earth radius.

$\underline{gp}$ is the absolute position of the user in polar coordinates. Let $\underline{wp}$ be the version of $\underline{gp}$ in cartesian coordinates. And let $\underline{up}$ be the user position in terms of Unity/HoloLens2 frame.

$\underline{up}$ is referred to $< u >$ which is a *left-handed reference frame*. This is due to the fact that Unity is mainly used to develop 2D games. In fact, left-handed reference frames are commonly used for cameras. Let's imagine for instance a object moving "forward" seen on the side: the movement is commonly perceived "forwards" when it is directed from the left to the right. Unity has axis "up" (the y axis), "left" (the x axis) and "look" (the z axis, pointing towards the camera is looking to).

The system will handle all the coordinates in left-handed reference frames, as a design choice. It is not the only way, obviously. In this implementation, both $< u >$ and $< w >$ will be left-handed reference frames.

3.2.1.1 Polar coordinates to Cartesian Absolute Coordinates

The first main problem to solve is to transform the polar coordinates to absolute coordinates; the cartesian absolute reference frame is assumed to be left-handed. Here are the formulas, from a simple geometrical reasoning. $\underline{gp} := (\phi, \lambda)$, $\underline{wp} := (wp_x, wp_y, wp_z)$, and R is the volumetric Earth radius, in meters:

$$\begin{cases} wp_x = R\cos(\phi)\cos(\lambda) \\ wp_y = R\sin(\phi) \\ wp_z = R\cos(\phi)\sin(\lambda) \end{cases} \tag{3.1}$$

The inverse procedure has to be developed with a bit more attention. The latitude ϕ can be easily obtained from the second equation as follows:

$$\begin{aligned} wp_y &= R\sin(\phi) && \text{from this} \\ \frac{wp_y}{R} &= \sin(\phi) \\ \phi &= \arcsin\left(\frac{wp_y}{R}\right) && \text{the final formula for } \lambda \end{aligned}$$

The longitude λ can be obtained combining the equations 1 and 3:

$$\begin{aligned}
\frac{wp_x}{\cos(\lambda)} &= R\cos(\phi) && \text{from the first equation} \\
\frac{wp_z}{\sin(\lambda)} &= R\cos(\phi) && \text{from the third equation, so} \\
\frac{wp_x}{\cos(\lambda)} &= \frac{wp_z}{\sin(\lambda)}
\end{aligned}$$

Which holds if and only if $\cos(\lambda)$, and $\sin(\lambda)$ are different from zero. So, the solution has to be studied depending on three cases.

Here's the solution for the first case, with $\cos(\lambda) \neq 0$, $\sin(\lambda) \neq 0$ and $wp_x \neq 0$:

$$\begin{cases} \phi = \arcsin\left(\frac{wp_y}{R}\right) \\ \lambda = \arctan\left(\frac{wp_z}{wp_x}\right) \end{cases} \tag{3.2}$$

In the case it holds $\cos(\lambda) = 0$, the points belong to the circle orthogonal to the Greenwitch meridian. With $\sin(\lambda) = 0$ the points belong to the circle passing through Greenwitch meridian. $wp_x = 0$ can be obtained also with $\cos(\phi) = 0$, which is a third singular case; in this case, the vector is aligned with the vertical axis from pole to pole, so it is impossible to determine the longitude λ.

3.2.1.2 Cross Product in left-handed reference systems

It is easy to prove that the formula of the *cross product* does not change using a left-handed reference system, by noticeing that the only difference between a left-handed frame and a right-handed one is the swap between y and z.

$$(\underline{u} \times \underline{v}) = \begin{cases} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{cases} \tag{3.3}$$

3.2.1.3 Transformations for left-handed reference systems

Transformation between two frames can be represented through a matrix operation taking the vector to transform as input and returning the vector projected in the new frame.
Let ${}^w_u T$ be the transformation matrix, and ${}^w_u R$ the orientation matrix; the transformation from frame $< u >$ to frame $< w >$ can be represented in this way:

$$w \begin{bmatrix} w_x \\ w_y \\ w_z \\ 1 \end{bmatrix} = \begin{bmatrix} {}^w_u R & {}^w_u O \\ 0_{3\times 3} & 1 \end{bmatrix} u \begin{bmatrix} u_x \\ u_y \\ u_z \\ 1 \end{bmatrix} \tag{3.4}$$

The relation is still valid also in left-handed reference frames in the same form.

3.2.1.4 Absolute to relative coordinates

Now that we have a rule to pass from polar absolute coordinates to cartesian absolute coordinates and viceversa, it's time to find a rule to project the Unity frame to the absolute one and viceversa. The basic idea is to find a matrix ${}^{w}_{u}T$ able to map the unity frame into the absolute one.

The Unity frame is created somewhere in the space when the device is started. To find a relation, a procedure can be developed, leveraging the fact that, in theory, for each point provided by the internal odometry, we can ask for a corresponding absolute localisation.

Which informations do we have? We have a first vector representing the position, which is the ray vector in absolute coordinates from the origin of the absolute frame towards the user's current position. We could fix a frame using that vector, normalized, as "up" axis. But this is not enough to define another frame: we need 2 vectors at least to build a frame. We can build it simply making the user move from one position to another: the motion creates another vector, with start point and end point known, that can be used as "look". Important to say that this vector is not necessairly orthogonal to the vertical axis choosen for this procedure, so we need another step to find the orthogonal projection in order to completely determine the transformation. The third axis "left" can be created as cross product between the two vectors created before.

Let's show the mathemathical procedure for getting the new frame from that idea. Let $\underline{gp}_1$ the first point from geolocation. The user goes forward, and the system obtains $\underline{gp}_2$. Both the points can be converted into cartesian absolute frames with the procedure described before, obtaining $\underline{wp}_1$ and $\underline{wp}_2$.

$$ {}^{w}\underline{\hat{wp}}_1 := \frac{\underline{wp}_1}{|\underline{wp}_1|} \qquad \text{the "up" unitary vector} \tag{3.5} $$

$$ {}^{w}\underline{\hat{r}} := \frac{\underline{wp}_2 - \underline{wp}_1}{|\underline{wp}_2 - \underline{wp}_1|} \qquad \text{it will be used for computing the "look" vector} \tag{3.6} $$

$$ {}^{w}\underline{\hat{r}}_{orth} = {}^{w}\underline{\hat{r}} - (\underline{\hat{wp}}_1 \cdot {}^{w}\underline{\hat{r}}) {}^{w}\underline{\hat{r}} \qquad \text{the "look" unitary vector} \tag{3.7} $$

The frame is completely determined by the base $< \underline{\hat{wp}}_1, {}^{w}\underline{\hat{r}}_{orth} >$. The third vector of the base can be found using a simple cross product:

$$ {}^{w}\underline{\hat{r}}_{left} = {}^{w}\underline{\hat{r}}_{orth} \times \underline{\hat{wp}}_1 \tag{3.8} $$

The last step is to compute the transformation between the two frames. The orientation matrix can be easily computed using the general formula which considers each axis projected to all the frames of the absolute frame. The first we can obtain diretly from the reasoning done before is the transformation from $< u >$ to $< w >$:

$$ {}^{w}_{u}R = \begin{bmatrix} {}^{w}\hat{\underline{r}}_{left} \cdot {}^{w}e_x & {}^{w}\hat{\underline{r}} \cdot {}^{w}e_x & {}^{w}\hat{\underline{r}}_{orth} \cdot {}^{w}e_x \\ {}^{w}\hat{\underline{r}}_{left} \cdot {}^{w}e_y & {}^{w}\hat{\underline{r}} \cdot {}^{w}e_y & {}^{w}\hat{\underline{r}}_{orth} \cdot {}^{w}e_y \\ {}^{w}\hat{\underline{r}}_{left} \cdot {}^{w}e_z & {}^{w}\hat{\underline{r}} \cdot {}^{w}e_z & {}^{w}\hat{\underline{r}}_{orth} \cdot {}^{w}e_z \end{bmatrix} \tag{3.9} $$

So that the entire transformation can be represented as described above.

$$ {}^{w}v = {}^{w}O + {}^{w}_{u}R^{u}(P - {}^{u}O) \qquad \text{direct transform} \tag{3.10} $$

$$ {}^{u}v = -{}^{u}_{w}R^{w}O + {}^{u}_{w}R^{w}(P - {}^{w}O) \qquad \text{inverse transform} \tag{3.11} $$

Now that we collected all the mathemathical tools to deal with the problem, it's time to present how the solution has been implemented.

3.2.2 Geolocalisation Solution Structure

The basic idea behind the implementation is to provide a component extracting the absolute position with a given period. Unity supports a easy-to-use integration with localisation, however, working only with android-based devices, so, in order to access the position, we have to interact directly with the Universal Windows Platforms API. To have a idea about the API call to use, snippet 3.2 shows the procedure required to extract absolute position using UWP in C#.

Figure 3.2: UWP Procedure to extract Geolocation

```
using Windows.Devices.Geolocation;

// HighAccuracy : BOOLEAN
Geolocator geolocator = new Geolocator
{
    DesiredAccuracy = (HighAccuracy ? PositionAccuracy.High : PositionAccuracy.Default)
};

Task<Geoposition> taskGeoPos = geolocator.GetGeopositionAsync().AsTask();
taskGeoPos.Wait(); // wait for a new position (distributed as Coroutine)

Geoposition pos = taskGeoPos.Result;
double latitude = point.Position.Point.Latitude;
double longitude = point.Position.Point.Longitude;
```

In order to make simpler the implementation of all the components requiring the position, it has been decided to provide a overlay between the high-level application and this base component extracting transformations. This component will contain also all the procedure to detect the transformation between absolute frame and relative frame. To compute this, a simple explicit calibration has been provided, as explained later.

Here the user is explicitly asked to enable the calibration. Another solution could have been to calibrate in background and then propagate the calibration informations to all the other positions already registered, in order to simplify the usage of the device.

3.2.2.1 Calibration Algorithm

The overlay on top of the base component provides also the calibration capability, structured in a way that it is possible to create a minimal coordination with the user. Here are the main steps of the calibration process:

1. The user is asked to stop in a place; the system tries to determine the average position over a certain number of measurements from the geolocation base component;
2. The user is asked to move forward until a certain distance;
3. The user is now still while the system detects the second position;
4. Now the system has all the elements to determine the transformation.

The process creates the transformation matrix, and from that moment it is possible to ask also the absolute position to the system, even offline.

3.2.3 Comments on Practical tests

System has been developed for testing purposes with all the blocks described above. Unity has a technical limitation due to the numerical precision in vector calculus: all the vectors and quaternions in Unity are encoded as float values, i.e. with a data type providing not enough numerical precision to deal with the computations for geolocation. It required to implement a custom library for vectorial computating in double format, supporting also transformations handling.

A system for collecting data has been implemented along with the absolute localisation components. Here are the data collected:

- Either data come from the absolute location or the relative one. In the first case, latitude and longitude are converted into absolute cartesian coordinates, and then in Unity coordinates; in the second one, latitude and longitude are computed offline using the last known transformation.
- Absolute coordinates in latitude and longitude
- Absolute cartesian coordinates
- Relative cartesian coordinates

Two sets of tests have been performed: indoor (using WiFi or also the phone hotspot) and outdoor (with phone hotspot only). Analysis of the results have been performed mainly offline with various tools, e.g. MATLAB. Some tests have been also filmed to recall the paths traveled during the analysis.

At level of internal odometry, results are satisfying in these cases: internal odometry is able to capture the shape of the paths during the practise tests.
However, results from absolute geolocation show a unacceptable lack of precision, even with a perfectly stable WiFi localisation. Coordinates are often far away from the real point: about 100 meters around the place of the test. And the movement does not reflect the movement of the user, especially in the first step, which is critical for a correct calibration. The result is a wrong orientation matrix and displacement.

From practical experiences, it comes out that HoloLens2 is not precise enough with the WiFi location. Moreover, the device neither supports GPS nor A-GPS, so the WiFi location, at native hardware level, is the only option we have to localise HoloLens2 through this kind of API.

3.2.3.1 Shifting point of view

The idea based on having one vector represented in both the reference frames at the same time is not applicable with such level of localisation precision, providing wrong results in terms of transformations. Scaling is not affected, so the shape of the paths is unchanged when data come from a offline transformation: the problem is related on the transformation only.

Trying to solve this problem, there are a number of possibilities to explore. The first one is basically a co-location between a system supporting GPS or A-GPS and HoloLens2. There are few old articles about this topic, nevertheless there are many discussions on the Internet: people need for localisation, and HoloLens2 does not support GPS localisation. The most common device supporting it is the phone; in particular, this kind of integration is particularly convenient since the Unity engine has a very good support for Android-based mobile applications, so one solution could be to use the phone as a beacon in communication with HoloLens2, building a mobile app able to provide precise position to HoloLens2. Other environmental sensors or wearable sensors can be applied as well.

Another interesting evaluated solution is to have only one position and then to find the orientation using the magnetometer integrated with HoloLens2, which is not easily accessible though. This does not solve entirely the problem since we need to know one point in the space in terms of both the reference systems. A magnetometer could solve the problem of finding a orientation of the frame, since using it we will have all the informations to choose a frame for the calibration. This solution has also the advantage to require less positions to the geolocation API, making a smaller possibility of errors since sometimes the initial position could be precise enough. Moreover, the orientation will be ofter correct at first, if the sensor is working fine.

The approach based on explicit geolocation can be developed in many ways for improving precision, but, under the scope of this work, another way has been choosen to solve the problem, as explained in a moment.

After all, in all the situations discussed before, the internal odometry produces correct measurements and correct proportions for the measurements. Absolute position can be corrected offline, as well as the orientation, trying to align a common map like the one provided by Bing or Google Maps, with the set of points generated by the HoloLens2 application, allowing to discover a map. The point is that initial position and orientation can be considered irrelevant for the proper functioning of the system.

To guarantee the interoperability among the systems involved in the rescue operations in terms of localisation and mapping, it is sufficient a *agreement*: before starting the operation, a real point in the space can be fixed, and used as "absolute frame" for every other devices and operators. Using this idea, we don't need a explicit absolute localisation, since it can be reconstructed offline by another process able to associate that origin to the correct geo-point. A device will make a initial error with respect to that frame when it starts work, which can be estimated asking the user to go in a certain position. Also the orientation error can be estimated in this way: the user is asked to calibrate the HoloLens2 device, for instance, staying in one position and looking towards somewhere during the calibration process. This will be the approach developed in this project.

3.3 Odometry-based Localisation System

This approach can be seen as completely opposite to the previous one: the main data come from the device itself, leaving the problem of coordinates conversion to further steps. This has the great advantage to be able to work completely offline.
Important to say that the previous IP-based solution have been developed for testing purpores only: once detected the precision problem, the approach has been changed to this one, passing through a number of intermediate development iterations.
The one explained in this section will include the problem of mapping as well, instead of the previous one that hasn't been developed much in this direction.

Accessibility graph model will be assumed, as formalised in the chapter 2: the places discovered by users can be stored in a graph structure built with the rules of that data structure.

3.3.1 Solution Structure

The HoloLens2 current implementation contains a set of components dedicated to the mapping of the environment as well as localisation inside the accessibility graph. The so called *PositionsDatabase* is a handle that interacts with the *Positions Database Low Level* that is the component dedicated to the storage and indexing of informations. The uppermost component provides convenient abstractions with respect to the low level, as well as automatic functionalities like, for instance, the creation of new positions during the exploration. Figure 7.3 gives a idea of the structure of the implementation.

The low level is meant to be a passive component with respect to the positions

database handle, shareable with other high-level services. Important to say that the low level is the most critical component of the system with respect to performances, since it allows to properly index and order positions. For this reason, it has been developed (as general guideline) to be replaceable with other components exposing the same interface. In any case, the complexity of the storage is hidden inside the low level.

3.3.2 The Positions Database

Main feature offered by this component is to provide a frame-by-frame update of the low level and to check distances from known points and the current position of the user in order to detect new positions.

Let's start discussing the update. Pseudocode 3 shows the implementation of the main cycle of the component, repeated at each frame.

Algorithm 3 Positions Database Update procedure

```
Require: mainCamera.transform                ▷ The user's current position
Require: lowLevelDB                          ▷ The posDB low level
  refp ← mainCamera.transform                ▷ Get the user's current position
  lowLevelDB.SortStep(refp)                  ▷ Update the low level index
  if checkReferenceDistance() then
    InsertPosition()                         ▷ Create new waypoint
    OnZoneCreated()                          ▷ EVENT: new waypoint discovered
  end if
  if current zone changed then
    OnZoneUpdated()                          ▷ EVENT: waypoint is changed
  end if
```

More will be said about the `checkReferenceDistance()` in the section describing how the component manages the creation of new positions. The last `IF` statement is implemented as a trivial inequality.

Notice that, to support more complex scenarios and testing implementations, the final implementation is able to read positions from whatever virtual object as well, not only from the internal odometry. The handle issues two types of events at the end of the update process:

- the `OnUpdate()` event, a signal issued when the position changes
- the `OnInsert()` event, issued when a new position is created

Due to a limitation of Unity in the implementation of the events, signals are issued as empty functions; the handle provides two public fields for retrieving the most recent informations (please refer to figure 7.7). A simple update is issued only once, whereas a insert operation is issued also as update, since the position changes verifying also the second condition. Position can change in any time, since the component checks the distance at each frame.

3.3.2.1 Localisation and Tracking Solution Design

Let's go deep by discussing the implementation of the current low level positions database. The main problem we're intested to solve is now the localisation of the user inside the accessibility graph. Part of the design phase has been developed during the previous chapter.

As usual, let's start from the simple concept. Each tiem the user moves, the system has to find the user's position inside the ones stored in the accessibility graph, and to take apart this information. All the positions are stored inside a list, which is kept always semi-ordered with respect to user's position. The main advantage for using lists is to be able to find a near position even if they are not directly connected.
But doing a complete sort at each frame is really a bad solution. A better one is to develop a algorithm able to efficiently distribute the update on many frames.

A first observation to point out is that we don't need the perfect order: for many features, it is necessary just to make the best effort to keep the list ordered depending on the current user's position.
The base idea is to make at most one operation per frame, scanning the entire array frame by frame, until the end, and then to repeat again. Does it make sense with respect to the application? Important to observe that, at each frame, the two arrays (as-is Vs to-be) are very similar with respect to the order criterion. If the user is walking (and, in our case, *carefully walking*) and we didn't set a extreme level of details on the path sampling, the user moves but the ordered array changes a little, so a limited number of moves will be necessary in this case for restoring the order of the array.

So, it makes sense a solution scanning the array making at most one swap at each frame. It will be always semi-ordered in the case the user is walking. Le's call the algorithm *Dynamic sort*.

3.3.2.2 Basic Version of Dynamic sort

Let's assume that HoloLens2 is able to provide a framerate high enough, with short processing time. From official documentation and practical tests (please refer to chapter 5), HoloLens2 provides a update frequency of 60 frames per second. This is a bit low for the application we have in mind, but let's start from the basic idea, making optimisations later.

The main operation will be the swap between the current element and the next one, which is performed or not, on the basis of the distance between the user and the waypoint. At most only one swap will be performed at each frame. Pseudocode 4 puts in details this idea.

This simple procedure is runned at each frame. It scans all the array, from the first element to the last one. If the element has distance greater than the following one (wrong order), then the two elements are swapped. Otherwise, the algorithm does nothing.

Algorithm 4 Dynamic Sort Simple

Require: P_{user} ▷ user's position
Require: dist(P_1, P_2) ▷ Distance between vectors
Require: db ▷ Array of waypoint references
Require: Swap(idx_1, idx_2) ▷ Swap of 1st with the 2nd
Require: idx ▷ The previously considered index, 0 to $N-1$

```
function SortStep(P_user)
    if idx ≥ len(db) then
        idx ← 0
    end if
    if dist(P_user, db[idx]) ≥ dist(P_user, db[idx + 1]) then
        Swap(idx, idx+1)
    end if
    idx ← idx + 1
end function
```

The main limitation of this idea comes out when the array has a "huge" number of elements: the index goes lost among a number of positions too far from the current one.
The current version of the project supports two simple types of optimisation: *clustering* (i.e. splitting the array into sections which can be separately updated) and *maximum number of indices* (i.e. a way to limit the number of updates inside the same frame due to clustering).

3.3.2.3 Cluster Optimisation

This is a "direct" method to solve the drawback of the too long list of positions to update: the solution consists in anticipating the update of far elements. The basic idea is to split the array into chunks of fixed length, that is declared as parameter of the algorithm. Inside each cluster, the update is performed using a dedicated index, with the same technique as in the simple version. The algorithm 5 explains how the update is performed in this case.

This optimisation anticipates the sort of far zones of the array at each cycle. The number of clusters increase each time a new element is added, increasing the number of updates per cycle as "side effect"; it could be a good trade-off if the length of the clusters is choosen cleverly.

In the final project however, it has been implemented a version of the algorithm which is slightly different from the one shown above: the algorithm has been implemented keeping a list of indices, one for each cluster. This solution has the advantage of making easier to compute the indices during the functioning of the system, but it is a less efficient solution, and pointlessy more complex to manage.

A limitation of this optimisation lies in the fact that the optimisation adds processing time to each step of the algorithm. Small cluster values affect badly the perfor-

Algorithm 5 Dynamic Sort Cluster

Require: P_{user} ▷ user's position
Require: dist(P_1, P_2) ▷ Distance between vectors
Require: db ▷ Array of waypoint references
Require: Swap(idx_1, idx_2) ▷ Swap of 1st with the 2nd
Require: idx ▷ The previously considered index, 0 to $N-1$
Require: $clusterLen \geq 3$ ▷ Cluster Length

```
function SortStep(P_user)
    if idx ≥ Minimun(len(db), clusterLen) then
        idx ← 0
    end if
    idx_w ← idx
    while idx_w < len(db) − 1 do
        if dist(P_user, db[idx_w]) ≥ dist(P_user, db[idx_w + 1]) then
            Swap(idx_w, idx_w + 1)
        end if
        idx_w ← (idx_w + clusterLen)
    end while
    idx ← idx + 1
end function
```

mances, but even choosing a large cluster length, when the points become a lot, the processing time per frame will decrease heavily the framerate of the device, which is unacceptable if we want to keep the device as most reactive as possible.

3.3.2.4 Max Index Optimisation

To solve the problem of to the increasing number of updates in the same frame, another simple optimisation has been provided: it consists of put a limit on the maximum number of updates in the same frame. This has been combined with the cluster optimisation: as the number of indices is below the threshold, the system keeps creating new clusters; and when the number of updates is greater than or equals to the maximum number of indices, the system starts summing one to the number of elements per cluster, increasing the length of all the clusters in order to keep the update distributed on the entire array.
The pseudocode 6 shows the implementation of this optimisation.

This pseudocode shows a version which is slightly different from the one implemented in the final project due to the different way of tracking indices: the current implementation uses a list of indexes which has to be entirely updated when the threshold is exceeded.

All the optimisation here proposed can be enabled or disabled depending on the situation: a interface is provided in Unity Enditor to tune the devices, depending on the desired performances.

Algorithm 6 Dynamic Sort Cluster Max Idx

Require: P_{user} ▷ user's position
Require: dist(P_1, P_2) ▷ Distance between vectors
Require: db ▷ Array of waypoint references
Require: Swap(idx_1, idx_2) ▷ Swap of 1st with the 2nd
Require: idx ▷ The previously considered index, 0 to $N-1$
Require: $clusterLen \geq 3$ ▷ Cluster Length
Require: $MaxIdx \geq 2$ ▷ Maximum number of updates in a frame

```
function SortStep(P_user)
    if len(db) >= MaxIdx · clusterLen then
        clusterLen ← Floor(len(db) / MaxIdx)
    end if
    if idx ≥ Minimun(len(db), clusterLen) then
        idx ← 0
    end if
    idx_w ← idx
    while idx_w < len(db) − 1 do
        if dist(P_user, db[idx_w]) ≥ dist(P_user, db[idx_w + 1]) then
            Swap(idx_w, idx_w + 1)
        end if
        idx_w ← (idx_w + clusterLen)
    end while
    idx ← idx + 1
end function
```

3.3.2.5 Creation of new positions

Now that we have a way to track the user's activity in terms of localisation, the next problem is to implement a check upon the position of the user in order to discover new waypoints in the space. In particular, here the basic idea is to detect a new waypoint when it is too much "far" from the nearest one with respect to the user. So, the algorithm will use the firt position of the list as the nearest point, and depending on the distance from it, the algorithm will decide if the current user's position is new or already known.

Let's assume that the positions database is the only using the low level. In the next chapter, other components will be provided able to insert new positions; let's ignore this kind of situations here.

Using this data model, the space is divided in "areas", each of them using one waypoint as center. The user is located in one waypoint when its position falls into the area around the waypoint, that is, in general, a sphere with a certain fixed radius (equal for all the waypoints).

The system tries to find the closest point to the current user's position, trying to find a area which that position falls into. What is such a waypoint cannot be found? The user discovered a new waypoint. But it is better not to add diretly the point.

The current implementation waits that the point is *at the double of the base distance* from the closest registered waypoint to register a new one. This has been decided to have a good distribution of all the points across the space; let's call d the parameter "base distance" that is the radius around the waypoint. The system considers also another parameter, which is the tolerance ϵ, so that a new point can be added when the user is at distance varying in the interval $[d-\epsilon, d+\epsilon]$. This causes a little of overlay depending on the choosen tolerance. Figure 3.3 gives a graphical idea of this rule.

There's a third parameter, available only in the HoloLens2 application and not in the server (currently), that is the vertical height of the point, introduced after some tests on the device to avoid the generation of a new point when the user moves down the head; the space around the waypoint is meant to be a cylinder with a given height, and a radius varying in the interval $[d-\epsilon, d+\epsilon]$.

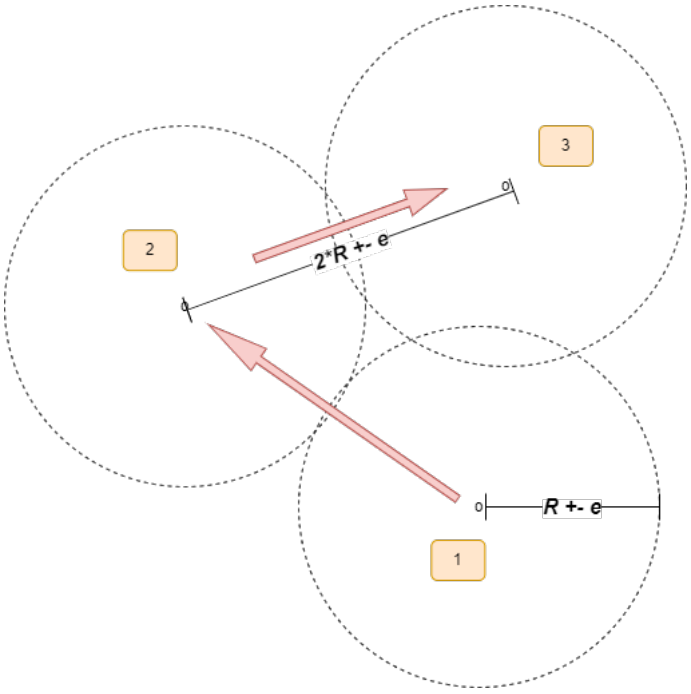


Figure 3.3: Representation of the distribution; the figure shows the reason why it has been chosen to take the double of the base distance: to have as less overlappings as possible between near zones.

3.3.3 Calibration Procedure Development

Calibration procedure has to be reviewed, compared to the one discussed for IP localisation.
The base idea is that rescuers can set a *operative frame* as a point on the floor, and a orientation by simply looking in a given direction. This allows to declare a frame which can be shared in the organisation. Notice that the frame can be one of a set of frames: it is sufficient to know how all the frames in the organisation are related each other.

From a mathematical perspective, the transformation can be handled in the same way as seen for the IP localisation, but reviewing a bit the concept.
In fact, in this case, it can be said that the device has a certain error in the positioning of the frame when the device is turned on. In terms of positions, this error is the distance between the operative frame and the Unity frame. In terms of orientation, the error can be estimated detecting a orientation matrix mapping the Unity coordinates into the operative frame. The transofrmation can be found quickly with a relation of this type: given ${}^{u}O_o$ the position of the user when the user starts the calibration with respect to the Unity frame $< u >$, a orientation matrix can be found, ${}^{u}_{o}R$, and the relation to frame $< o >$ is

$$ {}^{u}\underline{v} = {}^{u}O_o + {}^{u}_{o}R^{o}(P - O) \tag{3.12} $$

Unity detects a quaternion, which is another way to express orientation, similar to the orientation matrix; using the Unity API, the Unity frame sees the current frame croocked, so it detects the orientation in terms of ${}^{u}_{o}R$. This has to be inverted to find the correct relation. The following is the real direct relation, given ${}^{o}_{u}R$ as inverse orientation matrix:

$$ {}^{o}\underline{v} = {}^{o}_{u}R^{u}(P - O) - {}^{o}_{u}R^{u}O_o \tag{3.13} $$

With the last one, we're able to find the coordinates with respect to the operative frame.

The main advantage of this method of calibration is for sure the applicability of the Unity transformation API, since it is not required a extreme numerical precision differently from the other case.

3.3.3.1 Calibration Implementation

The implementation of the calibration involves two functional blocks:

- The first one, **Calibration Utility**, is a component providing methods for computing transformation as well as to coordinate calibration with the user experience;
- The second one, **Static Transform**, is a static object linked to the calibration utility, providing a convenient API for managing transformations at system level.

It has been implemented as a completely separated part of the system. It has been preferrable implementing the calibration in the StaticTransform, leaving the user experience only to the calibration utility, but it has not been possible since Unity avoids the access to the internal odometry for static objects, hence the object is required also for asking the current user location when a transformation is required.

Here, the calibration procedure follows these steps from the user's point of view:

1. the user turns on the device; a wrong frame is generated with respect to the operative one
2. the user goes into the referecence point and starts the calibration procedure
3. the system shows, for 10 seconds, a "crosshair" on the line of sight of the user
4. the user adjusts the orientation aligning the crosshair in the correct direction, and remaining still in that location
5. when time expires, a snapshot of position and orientation is maked by the system, and these informations are used for computing the transformation

Important thing to observe: the precision of the calibration depends on the user. A poor calibration could result in data skew, especially when the user is far from the reference origin.

To not overload the system, transformation has to be required by the single component. In general the Positions Database keeps working in the same way, handling the coordinates in Unity frame; transformations are used only when the system has to communicate with the other devices, or with a server as explained in the next chapter.

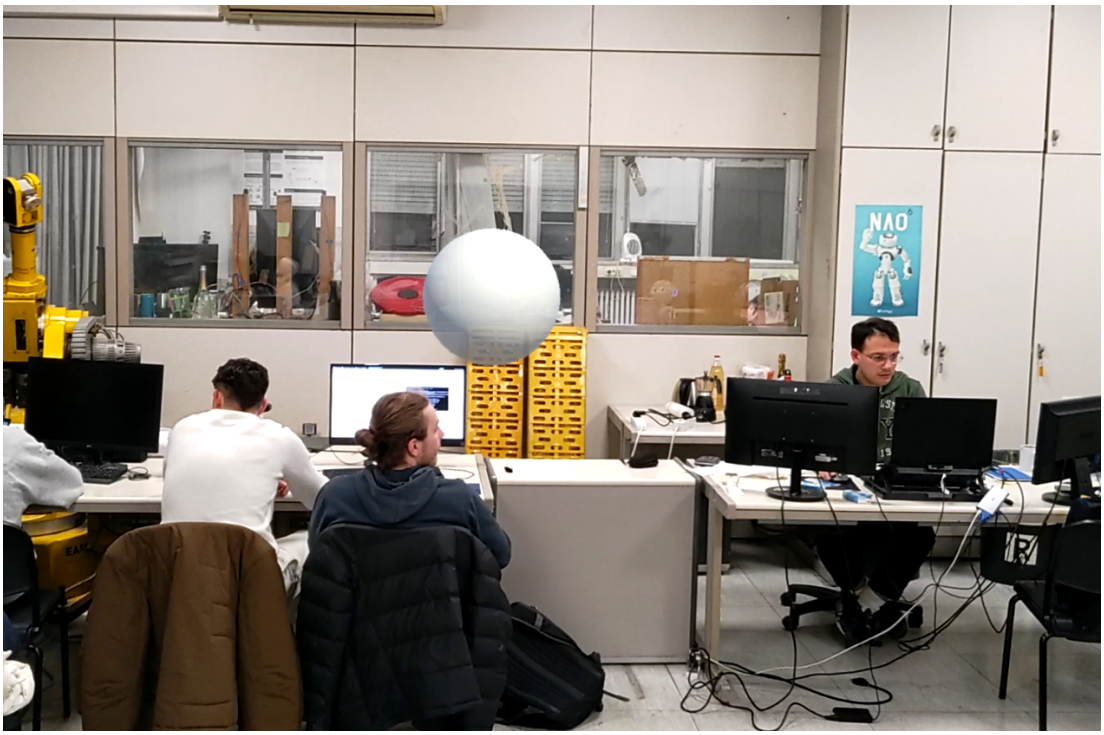


Figure 3.4: During the calibration process, a audio signal is emitted to warn the user that the calibration is about to start, and a "jerkling" cursor is visualised, aligned with the user's line of sight; it should be used as a aim for controlling the orientation. The user points the cursor to a given target, and keeps the same position until the calibration snapshot has not been collected.

Chapter 4

Cloud Integration

Summary

According to the initial point of view, making of a consistent knowledge base is strategical to coordinate a vast organization. This chapter shows how HoloLens2 application have been integrated with the server; the solution hereby explained can be extended to other devices, making a system which can collect mapping informations in a collaborative and efficient way.

4.1 Problem Statement and Motivations

Let's start from the problem statement. We're going to consider a more extended organization in which HoloLens2 is employed, made up of different types of agents. To give the idea, the organization can be composed by other HoloLens2 devices, but there are also autonomous devices such as terrestrain or aerial robots, and passive sensors placed in some areas to have informations about the environment, as well as other human operators in place (figure 1.3 gives the idea). Each of them can produce new informations to be shared with the other devices in real-time, mapping included.

Another important member of this vast scene is the central control unit, managing the rescue operations: for this part of the organization, it is very important to have a complete comprehension of the situation, as well as of the status of its agents.

A knowledge base is needed to coordinate such a organization, and to allow a collaboration among devices. In this case we're interested at providing a collaborative way of mapping a potentially unknown environment.
The approach here proposed is based on centralizing the informations in a single server managing the problem of integrating and spreading the informations across the agents in the area. This server will be accessible through a dedicated API and based on a relational database.

Let's discuss more specifically each of these requirements.

4.1.1 Connectivity in SAR applications

Connectivity is a complex topic to deal with in a true SaR application, since it is not guaranteed that in a disaster area internet connection is available. Moreover, it could happen that connectivity have never been present in that disaster area. We cannot take for granted connectivity.
Connections in SAR can be unstable, and with very limited bandwidth. A device in this environment have to be able to work autonomously for the 90% of the work time, and to take advantage of each connectivity spot. Using a classical approach based on a continuous stream of data is not the best choice for such a scenario. To ensure a good communication between a client and a server we have to reduce the weight of the connections and to optimize as much as possible the usage of the network.

Differently from the previous work, we decided here to not use Azure services, mainly due to the weight of the communications. Scene Observer Feature of HoloLens2 generates a lot of data: a complex, unstable data structure not directly informative. It is not easily portable to other devices, and it forces to send a lot of informations to the server, which have to be post-processed to extract the "main concept".

Communication in SaR appications for sure requires *simplicity*, since there's no space enough to manage complex communications using a unstable network. A data structure such as the accessibility graph can be exchanged easily, and it can provide useful informations without the need of post-processing the data in order to extract the basic informations. Instead, scene observer provides a too much complex representation which is not easily portable to other devices.

4.1.2 Security in SAR applications

Another fundamental requirement is *infrastructure security*; it is not a requirement concerning the robotics, but a important one to consider as basis of a shared system such as the one this thesis path wants to present. It cannot be excluded, especially in war scenarios or terrorist attacks for instance, that there could be somebody outside trying to hack the system to obtain malicious results, at least knocking out the operations.

A logon-based system has been implemented, a very common mechanism. Users have to login first, using personal credentials; after login, a user can ask to acquire some resources (devices) available in the organization. The last step is the logout. How to make it secure? Many concepts have been used in this probject; the most of them are based on intentionally complicating the procedures to make a potential attacker to spend much time in understanding how things work.

Another common mechanism is authorizations: a user is authorized to do something and not allowed to do other things. There are flags in the tables of the database stating what a user can and cannot do. If you don't have that authorization, you cannot ask for a device with these capables.

Temporary tokens are generated for most of the requests. A token means that the user can access to something to the system. Token is always sent by the client, but returned only one time by the server when the request is accepted. Tokens can be masked as well with fake tokens due to security reasons in th scope of the considered integration. Without the correct token, it is impossible to access, modify or release any resource.

And another mechanism is the clear classification of what a device can and cannot do, interms both of capabilities and authorizations. A device is another entity in the database, and it can have authorizations as well.

In the end, later on there are other algorithms focused to improve security during the functioning of the system, for example the fake token protocol. All these measurements make more complex to understand the situation for a external attacker unaware of how the system is configured or about the protocols. Even the error messages from the API are ambiguous, in order to not to provide useful informations for requests that are not the corrent or expected ones.

4.1.3 Collaborative mapping and Efficiency

The third fundamental requirement: such a system may be able to merge informations from many sources and to redistribute these informations efficiently when asked. More precisely, position and accessibility informations can come from many agents involved in mapping the environment. Each device can have informations not yet shared with the others, as well as other informations not yet aligned with the ones in the server.

The project implements a approach in which the server is always "passive" (mainly due to how HTTP works): it may be called always by the device when it is ready to exchange informations.
When a communication is required by the client, both the systems are aligned: the server integrates the missing informations putting them together with the other from the other devices, and sends "corrections" to the client, starting a nogotiation of informations aimed at syncing the information between the two systems.

The main problem, as consequence of this idea, is to solve the problem of positions alignment: in simple terms, given two sets of measurements, the first from the client, and the second from the server, we have to discard informations which are already present in the server. We can deal with it using a simple approach based on distances between points.

Another problem to overcome is about the storage. Using a relational database, which is the best way to store the informations keeping the system efficient? At least from a basic point of view, a big staging table is needed, collecting all the informations from many sources. But simply storing the measurement by just adding them in append mode will result in poor performance. We want that all these measurements talk amonth themselves, in a way such that redistributing the informations become as simpler as possible among devices. It results in the problem of creating a storage enabling the system to manage information distributions with efficient queries. Infor-

mations have to be available and shared in realtime immediately after the storage. This is the main reason why, in the scope of this thesis proposal, it has been preferred to work among a "dirty" representation of data, ordered enough to extract informations to redistribute as quickly as possible, postponing the generation of clean and final data to a process running in background.

4.2 The JSON Import-Export

Before starting with the final proposal, It is noteworthy to mention a first prototype built to check if the main idea can work. This has been developed just as an experiment, containing a good number of main idea behind the current implementation of the server part of this application. Moreover, it clearly shows some of the main difficulties they could arise in developing such a system.

The idea of the application is simple. Given the Positions Database as developed until now,

- The service will be capable of exporting informations from the database into a JSON file;
- After the export, in another session, data are imported from the same JSON file and the informations restored.

Unity has a bunch of useful utilities to handle JSON format, and that format is good for sending communications from the client to the API.

4.2.1 Solution Structure

The positions database export utility is the external service performing the import-export of the data.

For simplicity, the service is allowed to read directly from the positions database and from its low level, storing the waypoints and the paths. A storage hub (Figure 7.8) is the component in charge of reading from file or writing to file depending on the type of the request.

The component is triggered by two voice commands: "import" for importing data from JSON, and "export" to export data to JSON file. JSON Maker is a simple support class containing utilities to handle JSON transformations of objects.

4.2.2 Export Procedure

The export is pretty simple: just iterate over the points and collect all the measurements, and finally export them into the file in JSON format, using a simple graph traversing. Pseudocode 7 shows this functionality.

The first call creates a JSON structure collecting all the settings from the Positions database. Settings such as tuning are saved in the export. `dump` contains a list of

Algorithm 7 JSON Export Procedure

Require: JSONMarker ▷ cast wp/pt to JSON serializable objects

```
dump ← JSONMarker.ToJSONClass(PositionsDatabase)
for wp : Waypoint in PositionsDatabase do
    jsonWp ← JSONMarker.ToJSONClass(wp)
    for pt : Path in wp do
        if pt.startsFrom(wp) then
            jsonWp.paths.Add(JSONMarker.ToJSONClass(pt))
        end if
    end for
    dump.waypoints.Add(jsonWp)
end for

Export dump to JSON file
```

waypoints, which is empty at first and populated by the fisrt `for` cycle, which collects the paths and export them unter the waypoint belonging them.

The central `if` is very important: it takes only the paths "starting from" the waypoint considered in that moment. It ensures that the algorithm takes the paths only once without repetitions.

For simplicity, in this little experiment, change of reference frame is not handled. This means that the export is strictly referred to a given reference position, which is written on the file. System is not allowed to import the file if the operative reference frame is not the same in the calibration of the current session.

Export is performed with respect to a given ordering which depends on the current user's position. data will be ordered with respect the distance from the user.

4.2.3 Import Procedure

Import procedure is a little more cumbersone than export procedure:

- There could be *new informations on the device* with respect to the ones in the imported file.
- There could be positions already discovered in the knowledge base, meaning that some informations from the file are redundant.
- Points could be very distant from the first measured ones (*not solved in this implementation*).

The import algorithm should provide a mechanism able to compare and merge informations between the file and the kowledge base. Given the fact that the waypoints matching process can be done using a simple alignment based on distances, assuming

that the operative reference frame is the same as for the imported data, and assuming that the points are not too much far from the ones currently in the database, the algorithm 8 can perform the merge/import operation.

Algorithm 8 JSON Import Procedure

```
db ← reference to the Positions Database
jdb ← read from StorageHub and convert to JSON class
JSONMarker.FromJson(db, jdb)
wpRef ←jdb.CurrentPosition
db.lowLevel.sortReference ←wpRef
db.lowLevel.sortAll()
mergedLocations ← Empty Hashset of Waypoint
localKeyToWp ← Empty Dictionary from string to Waypoint
for jwp : JSONwaypoint in jdb do
    wp ← JSONMarker.FromJson(jwp)
    maxDist ← Distance(wp,wpRef)
    for dbwp : Waypoint in db.lowLevel.waypoints do
        if Distance(wp, dbwp) ≤ 2.0 * db.baseDistance - db.distanceTolerance then
            wp and dbwp are the same point
            localKeyToWp.Add(wp.Key, dbwp)
            mergedLocations.Add(dbwp)
        else if Distance(wp, dbwp) > maxDist or no more points into the db then
            wp is a new point
            localKeyToWp.Add(wp.Key, wp)
            db.lowLevel.waypoints.Add(wp)
            db.lowLevel.sortAll()
        end if
    end for
    for pt : JSONPath in jwp.Paths do
        wp1 ←localKeyToWp[pt.waypoint1]
        wp2 ←localKeyToWp[pt.waypoint2]
        if wp1 not linked with wp2 then
            create path between wp1 and wp2
        end if
    end for
end for
```

The first step is to order the low level list of positions relatively to the one registered immediately after the export. It is a simple form of optimization: just stop searching if the point in the positions database is too much far away from the one we're considering in the import. It allows to make more efficient the search.

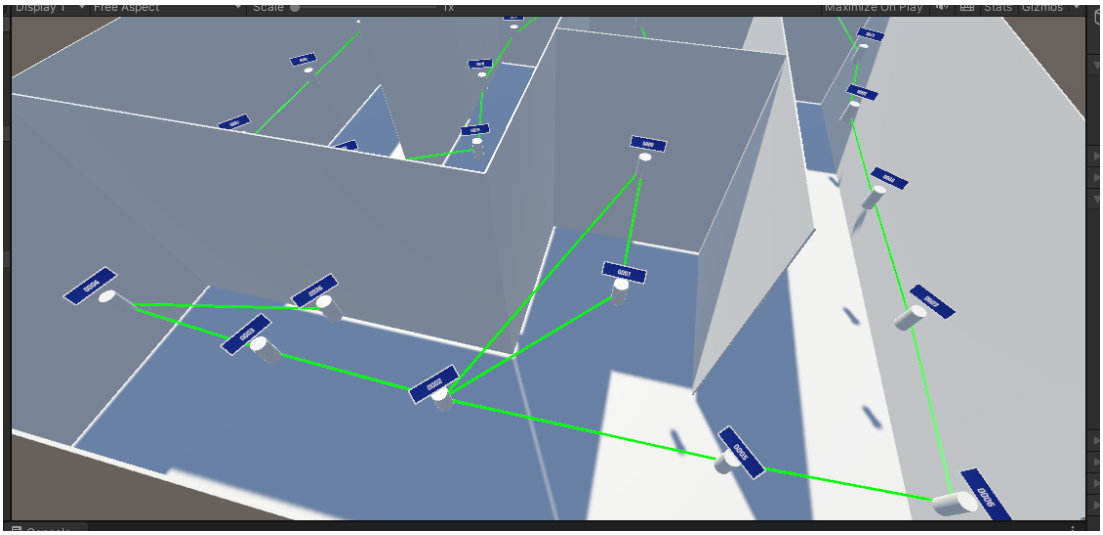
The maching between two points is performed executing again the common matching procedure: if the distance is lower than the threshold for the insertion, the points are the same, and there's no need to add another position to the positions database.

Otherwise, the new position is inserted at the end of the list and then the list is ordered (this step can be optimized just inserting in the right place the new node instead of re-ordering the array every time).

Edges are loaded after the waypoints, since, to perform connections, it is necessary to understand which waypoints are in the positions database and which nodes are matched with another alreaty present one. In simple words, I have to take the reference to both the waypoints to make a link between them, but the reference must be a waypoint inside the database.

There's another important reason to do this: the ID in the positions database could not be the same of the import. Let's consider for instance to have 24 points in the knowledge base, and 25 points inside the import. And let's assume that 13 points among the 25 of the import match perfectly with the knowledge base of 24 points. For these 13 points, it is not sure that the stable key is the same than the import: the alignment in this case prefers the information already present into the knowledge base, hence those 13 IDs have to be changed, and a similar thing happens with the remaining 12 points. The principle is that the ID have to be unique for each point, hence a change of IDs is required when new points are imported. This is a very important problem present also in the final application, and managed differently givng priority to the ID from the server instead of the one present in the client.

Figure 4.1: Anomaly found when importing data from a JSON; the algorithm gives some problems only while the user is walking during the import.



4.2.4 Conclusions and results

This little implementation has been tested following this procedure:

1. Start the simulation and perform a primary exploration

2. Export data on a JSON file
3. Stop the simulation
4. Start again the simulation
5. Explore a subset of the point discovered before
6. Import the points and check the "improved map"

The above mentioned algorithm gave good results: the system is able to merge the data with the previously measured positions. Sometimes, anomalies have been identified, such as the followings:

- **Cluster of nodes** due to a imperfect matching between two nodes in some particular case
- **Irregular connections** especially while the user is moving; sometimes it is possible that a link is created between two ndes which are too much far away each other. It is due to the second step of the algorithm.
- **Cyclical connections** : it is not a real problem in the end; sometimes the algorithm creates cyclic connections, but the system can handle correctly these cases

In conclusion, the algorithm have a good behaviour when the user is still during the import process. It is a drawback to handle in future development steps. But, except for these problems, the algorithm behaves in successful way during the tests, hence the base idea is robust enough to be extended to a more advanced level, i.e. using a shared server.

4.3 Solution Structure Overview

To implement the server, Docker Engine has been used. The server is a Docker Compose set of three microservices: the database (PostgreSQL), a container implementing the API (with FastAPI), and a proxy (NGINX), as shown in the schema below. 7.10 for a complete visual of the structure.

Noteworthy the use of virtual networks to deny every unauthorized access not passing through the proxy server: it allows to not to have the chance to connect directly to the database, making the whole structure more secure. The database can be inspected only connecting to the container hosting the server from inside the server, via command line. For the sake of simplicity, it has been decided to not create a second virtual network around the database, even if in a real application it is strongly suggested due to security reasons at first.

PostgreSQL is a relational database with support for geoqueries and with a modern SQL language. FastAPI is a lightweid framework to implement HTTP-based APIs,

easy to learn and to use. In this architecture, FastAPI is the only component allowed to directly perform queries on the database. In the end, NGINX is a fast proxy server able to manage a heavy flow of requests-responses, with many options to manage the traffic. For more information, please refer to Appendix 1 (7) containing a more technical explaination of the structure.

4.4 Data Model Overview

In simple words, there are two main entities: the **user**, representing the operator, and the **devices**. So far, the database considers only these two entities, plus a bunch of other ones specific for the HoloLens2 application; it can be extended to cope with a more complex scenario. Figure 4.2 shows a simplified diagram of the data model running on the database.
The current database can be divided into four servise areas:

- Users and users' access data, i.e. hashed passwords and tokens.
- Devices, and association between devices and users allowed to interact to them
- System activity tracking, mainly designed to track the access to the resources and the actions by the users.
- HoloLens2 integration, in particular the waypoints/paths tables; tables referring to reference positions and transformations could have a general usage in the database

In the following, the reader finds a short summary of tables in the data model and their meaning. All them are contained under a schema which is called `sar`. Only the really used ones have been considered.

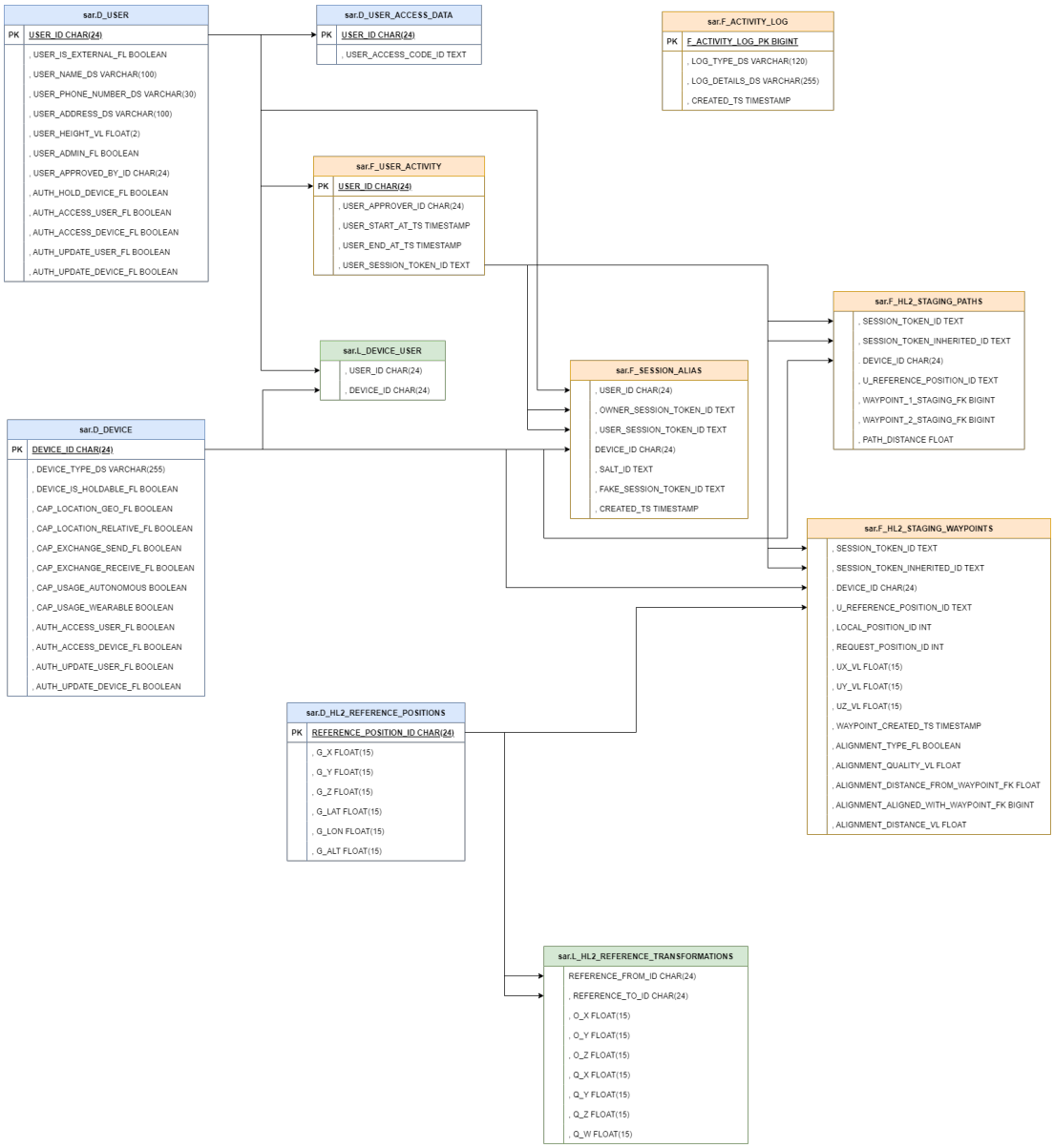
Tables storing the user's data, divided into access data and, anagraphical data.

- `sar.D_USER` : it describes the user, from the most general anagraphical informations (name, surname, phone number, address, ...), to the authorizations granted for that user (admin/not admin, write authorisations, read authorisations, either the user is allowed to hold a device ot nor). All the authorizations are encoded as simple flags in the table.
- `sar.D_USER_ACCESS_DATA` : it contains the access keysof the users, already hashed for security reasons.

Tables to track users' and devices activity, as well as system log.

- `sar.F_USER_ACTIVITY` : it allows to keep the history of the logins/logouts for a given user, as well as resources applied. The table also stores the session keys when generated during the login transaction. Still opened sessions will have the field `USER_ENDS_AT_TS` set as `NULL`.

Figure 4.2: A diagram showing the current data model, simplified. Arrows go from the main version of the field in one table, to the related field in another one table.



- `sar.F_ACTIVITY_LOG` : It is a general system log reporting all the activities of the system: loging, logours, resources acquired/released, with corresponding API call generating the event. The table reports also "suspicious" attempts to move inside the database, as well as taking track of failures. A minimal capability for detecting unsecure requests have been implemented; the table is managed mainly via API at the end of each transaction.

- `sar.F_SESSION_ALIAS` : there are some situations in which it is needed to hide a token with another token for security reasons. For further informations, please refer to the Appendix 2; the table simply maps one token in another generated one.

Tables storing devices informations:

- `sar.D_DEVICE` : it describes the devices, from the general characteristics (name, type, if the device can be holded by a user, ...) to a complete classification of the capabilities of the device, and to the authorization granted to that device.

- `sar.L_DEVICE_USER` : only some users can hold a (holdable) device; this lookup table links the user to the device and viceversa. If the user is linked to a device, it means that the user can hold that device if allowed.

And, in the end, here are the tables dedicated to HoloLens2 integration.

- `sar.F_HL2_STAGING_WAYPOINTS` : table of waypoints from the HoloLens2 devices

- `sar.F_HL2_STAGING_PATHS` : table of paths from the HoloLens2 devices

- `sar.D_HL2_REFERENCE_POSITIONS` : the database storing all the operative reference frames.

- `sar.L_HL2_REFERENCE_TRANSFORMATIONS` : it contains informations about how to map the coordinates from one reference position to another one in terms of position plus orientation.

4.4.1 Security Mechanisms Implementation

Access control has been developed for this project. This implementation of the API is designed for providing a fine access control for the resources, comprehensive of three control mechanisms: User's hierarchy, Items taxonomy, and items authorisations.

Users are divided into three categories: Admins (AKA Approvers users), Common users (AKA Approved Users), and external users. In general, to login, a user must provide two codes: its user code, and the user code of its approver. The approver must be logged in to approve a lower-level one.

- **Admin Users** : they have not the need to be approved by other users to login into the resources. Talkinga about credentials, admin users are *self-approved*: approver code is equal to the user code.
- **Approved Users** : they have access to the database, but only if the approver user is online; otherwise, the user is not allowed to access.
- **External Users** : they can ask data to the database,, but they are not alloed to acquire resources. This access profile is designed for users not belonging to the organisations, but only for data export when needed.

The system is able to track user's activity in the table `F_USER_ACTIVITY`, as well as to distinguish wrong calls by potentally security issues (it is currently a suggested feature). For instance, in the case the user tries to access in admin mode (approver user code equals to user code) without being admin, this request is likely to be not only wrong, but also potentially dangerous: it could mean, in the worst case, that a malitious user is trying to use one code collected in a not regular way (i.e. packet sniffing, or man in the middle attack), so it could indicate a brute force attack.

Items taxonomy consists of declaring what each device is able and is not able to do; this gives instruments to enable the system to check the semantic of a call. This holds only for devices.
For instance, a attacker could enter successfully the system, and create a application performing requests to the network. Important to point out that a user can do nothing without a device able to perform requests, so a attacker application has to use also a device ID to perform requests. A number of checks can be done; first of all, if the device is really capable to do what is asking to do: if not, actions could be taken against this user, and the irregular operation is notified.

A third level of protection is the *items authorisations*. It holds both for users and devices, since maybe the user can be allowed to do something, but the device maybe not, resulting in a not allowed request. Users can read or write data. Devices can read or write data. Users are allowed to hold a (holdable) device, such as a wearable device, hence also association between user ID and device ID is checked.

There's also a fourth level of protection, which is the *fake token protocol*, implemented at integration/protocols level, based on masking tokens with fake tokens. The reader can find more details in the Appendix 2 (8). And other security tricks are applied, for instance the fact to not share too much frequently tokens and sensible informations; access informations are provided once, assuming that all the applications have capability of memorizing and using correctly them.

In conclusion, the administrator has a deep control of what everyone and everything can or cannot do inside the system, so that it is possible to intercept malitious situations and having a insight at least in terms of frequency of irregular accesses.

4.5 The D.U. Protocol - Server side

After built a stable framework and provided patterns to implement API logics, more complex protocols and functionalities can be added to the application, adding new items to the main database, and new endpoints to the API, with proper transactions able to make everything work.
The main integration developed in this project is the *Download-Upload Protocol* (abbreviated, *D.U. Protocol*) which is a way to integrate HoloLens2 application, made up of two transactions able to integrate data and spread data across the organisation.

4.5.1 Protocol Design

It has been chosen, in this project, a *relational database* to store data structured as graphs (*semi-structured* data); other solutions would have been available, for instance a No-SQL based approach, e.g. using MongoDB or similar tools.
In relational databases, the simplest solution is to develop two tables for storing data: the first one, (`sar.F_HL2_STAGING_WAYPOINTS`), will store the waypoints; and the second one, (`sar.F_HL2_STAGING_PATHS`), will store the paths of the accessibility graph.

Let's introduce the metodology, starting from the most basic scenario. A idea is to store data in addition to the old ones each time a new measurement is generated. When download is required, data are sent back taking into account the user's position and a maximum distance. And, each time the device attempts to upload new data, informations are directly integrated.
HoloLens2 will have to *schedule* the two operations, alternating in a way that guarantees data to "agree" between client and server.
The main challenge is related to storage method, which has to be both efficient and reliable, and the hereby described storage method produces poor performances. Let's introduce step by step the approach implemented in this project.

The first challenge is to lower redundancy. In fact, the system could store new waypoints with no checks, most of them could be redundant since they are *near enough* to other ones previously recorded in the table; data should be organized so that they're ready to use with simple, efficient queries.
We could use a distance-based approach to filter redunant data away. When new points are uploaded, they have to be compared with the other data inside the table, and then added only if they're really new. To keep efficient such a *merging* operation, data could be structured to limit data amount during the matching phase: comparing data with a subset of the data is much better than comparing data with the entire table.
A important consequence is that being able to map positions into a set of known ones allows to *associate unique IDs to the positions*. **Comparing IDs is much efficient than comparing two vectors**. Each position can be identified by a *numerical identifier* which will have a *shared meaning* for the devices. We're referring to a shared knowledge base.

Let's say more. The first new measure (with respect to a threshold, using a distance-based data fusion) is the one which defines the ID. All the other measurements from all the other devices sufficiently near to that predefined one are discarded. This leads to a more conservative approach: the system tries to preserve the very first measurements, trying to *align* each other to a well-known one.
How to assign IDs? Hololens2 could *propose* new IDs to positions. It is a proposal indeed, since the device does not have visibility on the entire knowledge base, so it cannot infer either that proposai is already present in database or not.
As practical example, let's assume the device sends a `ID=41` (we're goiung to call it *request ID*) with position `(1.4, 0.09, 1.3)`, but the system already has `(1.5, 0.1, 1.5)` with has `ID=67` recorded. The consequence is that the position `ID=41` will be identified with the `ID=67`, and the server will return to the device that the position with request `ID=41` has actually `ID=67` (for "historical" reasons, we're going to label this new kind of ID as *local ID*). In the end, HoloLens2 will correct that ID from 41 to 67. This is **IDs Negotiation** and mainly affects the Upload phase.
HoloLens2 will contain a simple lookup table which is periodically compared and aligned with the database due to the negotiation mechanism. New measurements will be stored as temporary, waiting for the next negotiation with the server.
Using shared IDs, and with a good storage strategy, it becomes not relevant to say *which user has created the measurement*: it is enough to take *the first available one*, fact that allows to implement **collaborative mapping**.

To have the best performances, we could separe the simple storage from the data cleaning. In general, the current implementation works in terms of *best effort*, storing data in a "smart" way in order to use them without too much cleaning steps. Data, from generation, will be immediately accessible, allowing to reach the **realtime sharing**.
To be more clear, one limitation, intrinsic to the usage of a relational database, is that *SQL language does not support recursive queries*. Let's consider the situation in which the system first adds `(1,2)` and `(2,3)` with IDs 1,2,3. Later on, another user adds the `(1,3)` path. Since there's no support to recursive queries, the system cannot infer the redundancy of the new path with respect to the main data set, leading to a redundant table. The system has been implemente to execute all the transactions in only one query at time, in order to allow the DBMS to optimize operations, but sometimes it has been necessary to break queries, adding Python processing steps (tradeoff).

A general way to solve unhandled situations is to divide the information handling across two levels of processing:

- A *staging level* (under the scope of this project): it is a efficient storage able to do *the best effort* to clear data.
- A *quality level* (not in scope) containing clear and enriched data, obtained from

staging measurements.

Here are some of the situations which could be handled at wality level: *paths redundancy problem* (as said, SQL is not recursive, so composite paths are not handled at all); *transformations of coordinates* between one operative calibration and another one (staging stores data from different frames as completely different sets, in order to not make the process heavier, as well as to cope with eventually missing operative frame due to emergency management in time); *data skew* due to calibration errors, mainly not perfect first position or orientation (how much distance a device could cover with no need for a second calibration?), and others.

Calibration error deserves a bit of discussion. The initial positioning error in operative frame detection produces typically a kind of error almost completely filtered by the system if device and transactions are tuned at the same way. Errors *on zone border* are always difficult to manage: these happen when a point is aligned too much near to the border of another one. This leads to the need for **estimating the quality of data alignment**, quantified here in percent value. The system executes a approximation on the value from the client before recording it, and the quality of the selected type of approximation is computed by the system.
Orientation error could lead to a more severe displacement on the measures from the client, especially then the client is far from the initial position.
To make a quick estimation, let O_r be the origin of the (identified) operative reference frame, P_r the position, $\phi_{o,p}$ the angle between the correct orientation and the one used in the initial calibration. Considering that a orientation error produces a rotation of the map with respect to the one it should be, the displacement could vary at least with $|P_r - O_r| \sin(\phi_{o,p})$. Let's assume a distance of $1Km$ from the origin: assuming maximum error $\frac{\pi}{8}$ as example, minimum displacement is about 6 meters from the real position.
A correct error detection and resolution is almost unfeasible without a extensive data set. In order to be able to build a "quality" representation of mapping, it could be enabled a parallel continuous process which can read new data from staging table and analyse them to obtain the clean ones. This can be created in parallel with the API, as a component separatedly orchestrated at server level. This is the *data processor*, as described in chapter 1. This process could involve complex procedures to put in action.

4.5.2 Waypoint Staging Table

Snippet 4.3 shows the schema of the table `sar.F_HL2_STAGING_WAYPOINTS` as in the DDL. Fields not relevant for our discussion have been hidden.

The main information is the vector (`UX_VL`, `UY_VL`, `UZ_VL`), referred to a aprticular operative frame, `U_REFERENCE_POSITION_ID`. `REQUEST_POSITION_ID` refers to a waypoint ID proposed by the client during upload; `LOCAL_POSITION_ID`, under the scope of one dataset, represents a *shared identifier* for that position, obtained after the positions filtering/matching process.

Figure 4.3: Schema of Waypoints Staging Table

```
CREATE TABLE sar.F_HL2_STAGING_WAYPOINTS (
      DEVICE_ID CHAR(24) NOT NULL
      , SESSION_TOKEN_ID TEXT NOT NULL
      , SESSION_TOKEN_INHERITED_ID TEXT DEFAULT NULL

      , LOCAL_POSITION_ID INT NOT NULL
      , REQUEST_POSITION_ID INT NOT NULL DEFAULT -1

      , U_REFERENCE_POSITION_ID TEXT NOT NULL
      , UX_VL FLOAT(15) NOT NULL
      , UY_VL FLOAT(15) NOT NULL
      , UZ_VL FLOAT(15) NOT NULL

      -- alignment algorihm results
      , ALIGNMENT_ALIGNED_WITH_WAYPOINT_FK BIGINT DEFAULT NULL
      , ALIGNMENT_TYPE_FL BOOLEAN NOT NULL DEFAULT false
      , ALIGNMENT_QUALITY_VL FLOAT DEFAULT NULL
      , ALIGNMENT_DISTANCE_VL FLOAT DEFAULT NULL
      , ALIGNMENT_DISTANCE_FROM_WAYPOINT_FK FLOAT DEFAULT NULL
      );
```

Informations about user's session are tracked inside this table; here can be found the fields `SESSION_TOKEN_ID` and `SESSION_TOKEN_INHERITED_ID`.
Both tokens referred to user sessions: the first one is related to the user instancing the measurement; the second one could be null, or referring to another user. In fact, enforcing the observation of conservative approach, it is said that one user *inherits the session from another user.*
To be more clear, let's consider a empty staging table of waypoints. One device managed to connect: the system, at first, generates one record with `SESSION_TOKEN_ID` set as the current session token, and `SESSION_TOKEN_INHERITED_ID` with value `NULL`. Since it is the first request, and assuming to perform the first request immediately after the calibration, the new record will contain the coordinates *(0, 0, 0)*. It may be said that measurement is referred also to a particular `U_REFERENCE_POSITION_ID` in this implementation.
Now, let's assume that another client connects with the same `U_REFERENCE_POSITION_ID`. In this case, the server will store another row, with `SESSION_TOKEN_ID` properly populated, and setting `SESSION_TOKEN_INHERITED_ID` as the `SESSION_TOKEN_ID` of the previously registered client; in other words, this client *inherits points from another session.* All the measurements from the second client will be available also in the dataset defined by the session token `SESSION_TOKEN_INHERITED_ID` which will be the `SESSION_TOKEN_ID` of the first user. The same will happen for all the clients connecting later to the server. One client cannot belong to multiple datasets: when it checks in, it is assigned to a unique dataset depending on its operative reference frame. Inherited session token ID is choosen during the first request, and it must be used during the entire communication cycle.

In the following, we're going to use the term *dataset* to refer to data under a given session ID or inherited session token ID, depending on the situation. *Inherited session* will be the session ID which inherits points from another session, which will be called *inheritable*. *Check-in* will refer to the phase when a session is created after the first

calibration: a inherited session is assigned in this phase, and the assignment will be represented by the origin, which will be recorded inside the staging table of waypoints.

4.5.3 Support for Alignment Algorithm and Alignment Quality Heuristic

Each time a device tries to perform a upload, data have to be compared to the other ones in the same dataset before storage. Data will be recorded as *completely new* if there's no data *close enough* to the uploaded ones; negotiation process would procude new shared IDs. Upload of already known data causes a simple renaming of positions.

As in DDL 4.3, inside `sar.F_HL2_STAGING_WAYPOINTS`, a number of columns starting with the prefix `ALIGNMENT_` are included, with the following meanings:

- `ALIGNMENT_DISTANCE_FROM_WAYPOINT_FK` is the key referring to the one nearest to this one at the time when its insertion has been evaluated.
- `ALIGNMENT_ALIGNED_WITH_WAYPOINT_FK` is `NULL` in case of no match, otherwise it is equals to the `ALIGNMENT_DISTANCE_FROM_WAYPOINT_FK`, i.e. the point that matches with the current one.
- `ALIGNMENT_DISTANCE_VL` is the distance between the current point and the point taken as reference by the algorithm at the time the point have been uploaded.
- `ALIGNMENT_TYPE_FL` indicates wether the point have been matched with another one (`TRUE`, i.e. is redundant) or it is a new point (`FALSE`).
- `ALIGNMENT_QUALITY_VL` is a percent value indicating the *quality of the decision* in `ALIGNMENT_TYPE_FL`.

Most of these fields listed above allow developers and operators to understand the reason why some measures have been approximated to other ones.
`ALIGNMENT_QUALITY_VL` allows to evaluate in percentage the quality of a match or a non-match. The principle is that, when the distance is beyond a given threshold, the algorithm creates a new local ID; otherwise, it identifies a match, hence a redundancy. This kind of behaviour could work badly sometimes, especially when the condition is satisfied at the border of the radius around a waypoint. The match is always evaluated with respect to the nearest point available in the dataset.

If two points match, the quality is evaluated in this way. Let r be the distance between the points, D the threshold distance; a simple linear relation is given, lowering the value from 100% (perfect match, zero distance) to 0% (worst match, too much near to the border).

$$\texttt{ALIGNMENT\_QUALITY\_VL} = \frac{D - r}{D}$$

For the contrary case $r \geq D$, the system performs a creation of new local ID with a quality following this formula:

$$\texttt{ALIGNMENT\_QUALITY\_VL} = 1 - e^{b \cdot (D-r)}$$

In the second case, exponential trend has been choosen because of its monotonical behaviour towards its limit. Figure 4.4 represents the trend of the heuristic: the zone before the zero evaluates the quality of the matching, whereas the side after the zero represents the non-matching case (quality increases with the distance from the border). To have a optimal tuning with respect to base distance set in the devices, zero should fall at the border of the waypoints; in the example of the figure, base distance is precisely one meter from the center position.

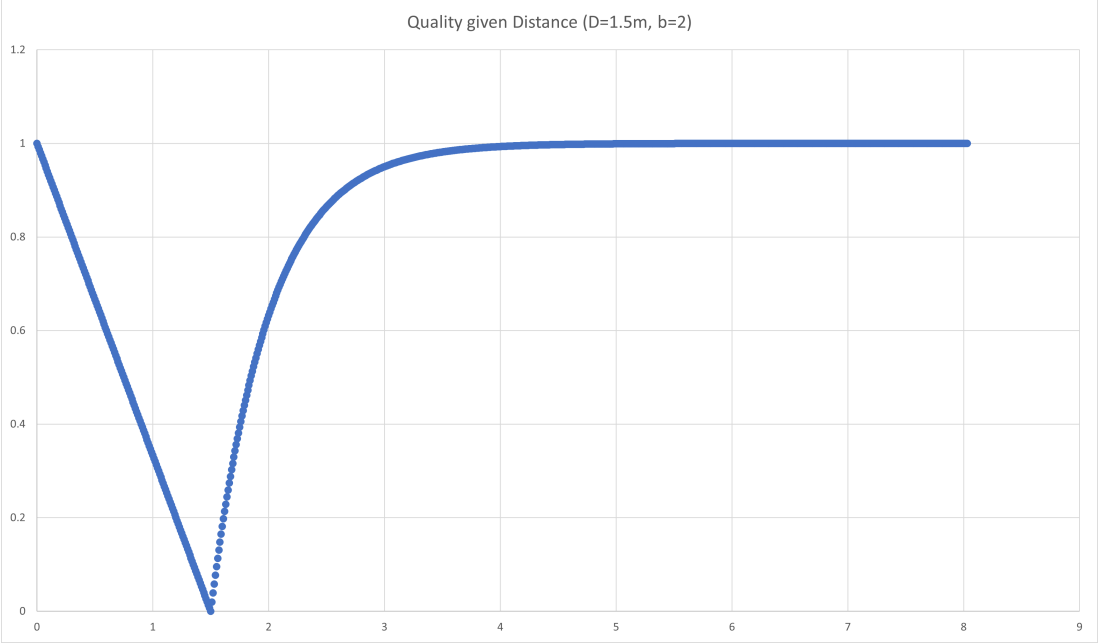


Figure 4.4: Maching Quality Heuristic, in the case of D=1.5m and b=2; it can be used to evaluate the quality of a matching of one uploaded point with another previously recorded one.

4.5.4 Paths Staging Table

Snippet 4.5 shows the schema of the table of paths `sar.F_HL2_STAGING_PATHS` as in the DDL.

Most of the things explained for the waypoints still hold also for paths. Storage logic is pretty much the same: it is based on session token IDs and inherited session token.
In this table, each row contains a path, represented as linkage of two waypoints referred through their *surrogate keys* defined in the master table of the waypoints.

The current implementation uses the paths table as slave table, preferring to handle

Figure 4.5: Schema of Paths Staging Table

```
CREATE TABLE sar.F_HL2_STAGING_PATHS (
DEVICE_ID CHAR(24) NOT NULL
, SESSION_TOKEN_ID TEXT NOT NULL
, SESSION_TOKEN_INHERITED_ID TEXT DEFAULT NULL
, U_REFERENCE_POSITION_ID TEXT NOT NULL
, WAYPOINT_1_STAGING_FK BIGINT
, WAYPOINT_2_STAGING_FK BIGINT
);
```

the waypoints table first. This is the main reason why this table has not developed too much with respect to the one containing waypoints.

4.5.5 Server Protocol Expected Usage

In appendix 2 (8), the reader can find more information about the way the two endpoints, *Download*, and *Upload*, have been implemented in the current version of the project. Here's how these endpoints are supposed to be used by the client.

The client has to make the first request immediately after the calibration. In other words, the server assumes that the first API call is issued while the client is located exactly in the origin of the reference system declared.
The **download** has been chosen as first request. A reason to prefer download instead of upload as first call is the fact that the client could not to have new informations to upload at the very beginning of the working cycle: it it more likely that the first really useful call would be the download.
Another reason is that client and server have to be aligned at first. Local IDs have to match with the shared IDs of the client, to be sure that both the systems know what is already said and what not.

After this process, the user can call download and upload in a unspecified order until it doesn't log out to that resource. Download will return only informations which are really new for the client, and inside a given area, even coming from different devices attached to the same dataset.
Upload will compare request informations with the ones under the dataset, and will return a set of alignments that the client must use to align its shared IDs to the ones inside the shared dataset.

4.6 The D.U. Protocol - HoloLens2 Integration

At HoloLens2 application level, integration is designed "on top of the stack" with the base implementation of the positions database. A new package, able to manage the details of the networking, has been developed, as well as a new service inside the positions database package able to use the netowrking package to communicate with the

server.

Positions Database has been designed to support new features and integrations in addition. To implement this new feature, another class has been developed, *PositionsDatabaseClientUtility*, which uses both the Positions Database low level and the controller. The access to low level is required for performing the updates, whereas the access to the controller is required to retrieve new positions when a new one is created. Please refer to figures 7.2 and 7.9 to have a idea. The entire integration runs in background.

The new class inside the Positions Database package requires another component named *SarClient* offering all the functionalities to deal with the details of the DU protocol, in a way such that the "orchestrator component" (the client utility) has just to call the methods and to coordinate the update processes.
SarClient contains a number of features. First of all, it provides abstraction with respect to the exact method for accessing resources through credentials. It can manage all the tokens needed for communication, and it can provide a simple interface for login-logout processes where the upper class has just to call the operation, to wait for it, and to check either the operation succeeded or not at the end of that operation. This class also needs to access the two main blocks of the positions database to perform operations. In particular, this component is able also to update database, so the orchestrator class has only to "schedule" operations, and nothing more.

SarClient is the top of a stack. *SarAPI* is a static class containing the methods than really carry out the API calls, asking to the server and obtaining and adapting response to be integrated with the database. *SarAPI* is focused on transferring data; *SarClient* handles data and integrates them in the positions database. *SarAPI* is also a static class, whereas *SarClient* is a component. This means that there could be many clients, but only one API handle for that system. It makes sense since the the connection must be unique for the device: one device cannot open different sessions with the same server. Concurrency is handled by the *SarAPI* component, but the possibility to have different clients upon the same API handle is not currently exploited.

4.6.1 Shared Keys and Stable Keys

Many points inside the HoloLens2 application make use of unique Keys to identify a position. For instance, components such as the *PathDrawer* need to have keys for identifying a position to manage the storage in efficient way. Moreover, assigning IDs to the position is of primary importante to identify a position and to share it with the server without having to work every time with distance or other complicated methods. In this implementation, a simple integer key is employed, and new keys are generated as ascending sequence.

The solution proposed in this work is to create two independent keys:

- **Stable Keys** are generated by HoloLens2. This is a strictly monotonical index from zero, and it cannot change.
- **Shared Keys** are the ones aligned with the server

Stable keys have internal meaning; they are never shared with the server, whereas stable keys can change in any time. The system tries to keep them always updated.
In the first phases of the project, keys have been not considered as shared between device and server, but only used for managing internal storage for building visuals. The consequence is that the first version of the application works as the HoloLens2 device is the only one in the entire organization, which is wrong when we introduce a server side able to merge measures from different devices; meaning of IDs have to be reviewed.

There is a complication when a update is performed. IDs can change, since the server can review them, and HoloLens2 have to align its internal knowledge base to the server. But visuals work on IDs which are (wrongly) assumed *unchangable* at runtime. In case of change of some ID, all the storage managers have to be updated, slowing down the integration of new informations.
In general this approach doesn't work so badly; but it would be better to work with one key; a next iteration of the project could aim to use only one key instead of two, maybe changing a little the way the visuals are managed by the system. Also the positions database low level has to be reviewed to use only one key. Having two keys makes more comples the management of the resources, and enforces to implement a bunch of useless mappings around all the application. Adding too much levels of indexing slows down the performances of the application.

The usage of shared keys along with the stable keys "in addition" lead to a problem on the updates of keys for instance. Let's consider that, at the basis, the positions database is basically a list. Stable key does not change, but shared key can, hence the application has to manage a search in the entire list for keeping them update, which is unacceptable with a huge amount of points: it would severely affect performances.
To avoid such a situation, it has been implemented a internal lookup table (i.e. a dictionary) at the database low level: it can be fully managed by an external plugin. When a shared key changes, it is enough to change just one record inside this table instead of searching the instance of the point inside the database list. It can be considered as another level of indexing upon the storage: this lookup table will point directly to the instance.
Currently, search given shared key is supported, since it is the best solution for the implementation of the data integration plugin.

4.6.2 High Level Protocol Orchestration

The client has to execute a first "handshake" before asking for the first download. The component *SarClient* asks first for the status of the service; when the service is online, it logs in with user credentials and then with device credentials.

After the connection is established, the device waits fro the calibration before the first download because the server assumes that, in making that API call, the client is at the origin of its frame. After that, the main working cycle can start in background, silently integrating informations into the system.

Let the `SarNet` be the main handle for connection, i.e. the object containing all the coroutines needed to perform the API operations in details. Algorithm 9 gives a idea of how the integration orchestration proceeds.

Algorithm 9 HoloLens2 D.U. Protocol Orchestration

```
if not SarNet.online then
    break                                          ▷ Server is offline, retry later
end if
await SarNet.Connect()
while calibration have not been done do
    await next frame
end while
await SarNet.Download()                                          ▷ first download
remainingTimeDownload ← Download time param
remainingTimeUpload ← Upload time param
while True do
    frameTime ← get time from the previous frame
    remainingTimeDownload ← minus frameTime
    remainingTimeUpload ← minus frameTime
    if remainingTimeDownload ≤ 0 then
        await SarNet.Upload()
        await SarNet.Download()
        remainingTimeDownload ← Download time param
    else if remainingTimeUpload ≤ 0 then
        await SarNet.Upload()
        remainingTimeUpload ← Upload time param
    end if
    await next frame
end while
```

Procedure 9 can be tuned using download period and upload period. Usually, upload transaction excange the most of informations, and, as seen in the server section, tries to keep HoloLens2 dataset aligned with the one inside the shared dataset on the server. To not to have too much informations to exchange in one shot, it is better to upload frequently, using the download with a longer period (less frequeny).
This is the suggested tuning, but also other strategies can be used. For instance, forcing the download to be more frequent instead of upload, if there are other sources able to produce new informations more frequently. The protocol allows to exchange informations to realize different alignment modalities combining the two basic operations.

Upload is called immediately before the download, in order to fill the wholes in the data into the device. Executing the upload before the download is a method to prepare the dataset before new data are integrated: in this way, the device can avoid to perfom more swaps than required.

During the exploration, the component takes apart a queue of references to the new positions inserted, collected each time the positions database notifies a new position discovered (i.e. by event). These positions are not gearanteed to have the ID the system is assigning to them: they need to be renamed or approved by the server: the system proposes a shared key when the point is inserted (a stable key is automatically assigned too). When it's time to upload, the queue is processed and passed to the *SarClient* to be exchanged with the server.

In the end, the solution to orchestrate the alignment is based on the time in this implementation. Other methods are feasible: for example, it could be tested another approach combining time and maximum number of items inside the queue. For instance, when the number of items is greater than a given threshold, this triggers the data exchange. Moreover, this approach can be reviewed in order to implement a maximum number of items exchanged in one transaction, splitting the communication into chunks of uploads. This last type of update in particular is a bit more challenging, since it is necessary to equip HoloLens2 with a implementation able to resolve alignment of points which are discovered but also received from the server: a not negotiated point becomes known in that case, making pointless to send it back again in the next upload.

4.6.3 Download Implementation

Upload and download are implemented in the HoloLens2 application as separate sets of methods. Here below the basic steps of the download call:

- The orchestrator calls the low level (the SAR Client), waits for the result, and then integrates the results in the positions database, managing the problem of positions generated while in the time the server spends to respond.
- The SaR Client is basically a bridge between the positions database and the server, marshalling data from one form to another one to make support for efficient elaboration when data integration is executed. PosDB uses data in custom classes, whereas SarAPI uses JSON serializable classes.
- SarAPI performs the server call, serializing data and exchanging them with the other side. This class automatically assigns the right credentials for communication, handling all the details under the hood.

This is the basic idea. Now, let's go deep into the algorithms used for performing these operations.

First of all, immediately after the static object returned the results from the server, results are elaborated to extract the real new results with respect to the informations

at disposal, logging and deleting potentially anomalies. The class keep tracks of the IDs exchanged during the entire communication process in a dictionary, which we can call `archiveWps`, associating a integer (the local ID) to the JSON class waypoint. This step can be considered as a pre-filter of the results from the server, to endure that there are no problems during the communication. It exists also a dictionary `archivePts` containing paths and with a similar function.

The first step is to make a selection of waypoints among the ones received from the server. Then, paths are collected. Algorithm 10 details the procedure.

Algorithm 10 Data Collection and PreFiltering on SarClient

```
UpdatedEntriesWps ← list()                          ▷ Public for the class
for all wp : waypoint from server response do
    if wp.localID not yet inside archiveWps then
        tup ← (wp.localID,wp.localID,wp)
        UpdatedEntriesWps.Add(tup)
        archivedWps[wp.localID] ← wp
    else
        continue                    ▷ the point received is redundant (anomaly)
    end if
end for
UpdatedEntriesPts ← list()                          ▷ Public for the class
for all pt : path from the server response do
    k12 ← (pt.wp1LocalId, pt.wp2LocalId)
    k21 ← (pt.wp2LocalId, pt.wp1LocalId)
    if archivePts already contains k12 or k21 then
        continue                    ▷ received redundant path (anomaly)
    end if
    tup ← (k12,k12,pt)
    UpdatedEntriesPts.Add(tup)
    archivePts.Add(k12)
    archivePts.Add(k21)
end for
```

The complex format of the two lists is due to the fact that they are used also for renaming from uploads. Of course, download transaction does not need rename, therefore IDs in the tuples are equals each other. The item in the list only carries the information to integrate, and not a renaming.

The second part of the job is the data integration, performed by the "plugin" positions database client utility. Please refer to procedure 11.
The low level database contains also a lookup table containing the association between the local ID and the instance of the stored waypoint. Each insert operation will create a new point and a new record inside this lookup, so the procedure to integrate data must

take into account that new positions could have been generated during the execution of the transaction, and their IDs could not be correct with respect to the received informations. Here comes the max ID provided by the server: that number means that there is no local ID in the server which is not greater than it. Hence, the solution applied here is to swap elements with wrong ID from the initial position to another one which is greater than max ID.

Here's the algorithm:

Algorithm 11 PosDb Data Integration from Download

Require: $UpdatedEntriesWps$ ▷ updated waypoints from the server
Require: $UpdatedEntriesPts$ ▷ updated paths from the server
Require: $PosDB$ ▷ the positions database
Require: $PosDB.lookup$ ▷ association local ID to waypoint instance
Require: $PosDB.lowlevel$ ▷ the low level of positions database
Require: $PosDB.lowlevel.db$ ▷ the instances DB at low level
Require: $MaxIdx$ ▷ maxID from the server

```
wpIndex ← MaxOf(MaxIdx, len(PosDB.lowLevel.db))
for all item in UpdatedEntriesWps do
    wp ←item[2]                                   ▷ take the waypoint
    localID ←item[1]                              ▷ take the local ID of the wp
    if PosDB.lookup contains wp then
        wpIndex ← wpIndex + 1
        PosDB.lookup[wpIndex] ← PosDB.lookup[localID]
        PosDB.lookup[localID] ← wp
    else
        PosDB.lookup.Add(wp)
    end if
end for
for all item in UpdatedEntriesPts do
    wp1id ←item[2].wp1LocalId
    wp2id ←item[2].wp2LocalId
    if either wp1id or wp2 not found in PosDB.lookup keys then
        continue                                  ▷ received a unknown path!
    end if
    PosDB.lowlevel[wp1].CreatePath(PosDB.lowlevel[wp2])
end for
```

4.6.4 Upload Implementation

The upload transaction is implemented in a way similar to the download one as a stack of methods, as summarized below:

- The orchestrator calls the low level (the SAR Client), waits for the result, and

integrates the renamings inside the low level.

- The SaR Client calls the method passing new positions to it, then it writes the renamings in public fields, offering them to the orchestrator
- SarAPI performs the server call, serializing data and exchanging them with the other side, as before, calling the Download API endpoint

Let's follow the flow upwards. The Sar Client receives the lists of waypoints and paths to send to the server, already formatted in a JSON serializable format. This component has also to update its internal archive for result filtering. Procedure 12 is performed immediately after the API call.

Algorithm 12 Data Upload on SarClient

Require: $uploadListWp$ ▷ positions uploaded from the client
  $uploadListPt \leftarrow$ list() ▷ Not needed here
Require: $renamingLookup$ ▷ renamings from the server as dictionary
  $UpdatedEntriesWps \leftarrow list()$ ▷ Public for the class
  for all wp : waypoint in $uploadListWp$ **do**
    $UpdatedEntriesPts \leftarrow list()$ ▷ Public for the class
    $finalLocalID \leftarrow$ renamingLookup[wp.localID]
    if wp.localID $\neq$ finalLocalID **then** ▷ renaming
      $tup \leftarrow$ (wp.$localID$,finalLocalID,wp)
      $UpdatedEntriesWps$.Add(tup)
    end if
    if wp.localID $\notin$ archivedWps **then** ▷ archive
      $archivedWps[finalLocalID] \leftarrow wp$
    end if
  end for

About the paths, notice that it is sufficient to rename waypoints to make the paths automatically changing: the structure implemented to store graph is capable of taking IDs directly from the linked waypoints.

Now, all these infomations have to be integrated to the dataset. 13 is the procedure implemented at level of positions database client utility.

Algorithm 13 PosDb Renamings Integration

Require: $UpdatedEntriesWps$ ▷ updated waypoints from the server
Require: $PosDB.lookup$ ▷ association local ID to waypoint instance
Require: $MaxIdx$ ▷ maxID from the server
  $wpIndex \leftarrow MaxOf(MaxIdx, len(PosDB.lowLevel.db))$
  for all *item* in $UpdatedEntriesWps$ **do**
    $idOld \leftarrow$item[0] ▷ the old local ID
    $idNew \leftarrow$item[1] ▷ the new local ID
    if $PosDB.lookup$ contains $idNew$ **then**
      $idx \leftarrow wpIndex$ ▷ *what if this location is not null?*
      $wpIndex \leftarrow wpIndex + 1$
      PosDB.lookup[idNew].localID ← idx
      PosDB.lookup[idx] ← PosDB.lookup[idNew]
      PosDB.lookup[idNew] ← $NULL$
    end if
    PosDB.lookup[idNew] ← PosDB.lookup[idOld]
    PosDB.lookup[idOld] ← $NULL$
  end for
  Properly set local max index from $wpIndex$

Chapter 5

Testing of the final Project and Results

Summary

At the end of this long journey, the last step is focused on performances evaluation. In the following sections, it will been applied a evaluation approach based on *key performances indicators* (KPIs): after designed a number of statistics quantitatively describing how the system is working under the hood, tests are done reproducing particular scenarios and trying to find the best settings from the application.

In the following, all the KPIs are defined, and for each of them, interpretation and properties will be provided. Then, they will be introduced the types of tests developed for testing the application; each test will be analysed using the known KPIs.

5.1 Main Key Performance Indicators

A key performance indicator is a measure of a quantity which have a particular meaning related to the problem we want to study. Each of them has particular properties and a particular way to evolve; they are related each other, and some of them have a bit of limitations.

Part of the project is implemented to extract statistics in CSV files, called *reports* in the following sections. KPIs are collected mainly in two reports:

- **Chasing Speed Metrics**: it allows a precise evaluation of the performances of the positions database, collecting different KPIs from that set of components in a way that allows to take into account also the relation between them. It refers to the "ability" of the system to "chase" the user's position inside the mapping data structure.

- **Client Stats**: it collects the main statistics about the connection of the client with the server; it provides details about the amount of data exhanged, both in

general terms (KiloBytes) as well as with more specific informations (e.g. the number of waypoints exchanged).

Also other small reports have been developed, for instance the one measuring the frame rate only, but the ones previously cited are the most relevant. All the reports can be easily analized offline with a common spreasheet application such as Microsoft Excel or Google Sheets.

The following is not a exaustive list: many other metrics have been implemented to assess how the application is working.

5.1.1 KPI Frame Rate

This is the very first KPI one person can come up with to test the performances of the system. It is defined as the number of `Update()` calls per second. There are many ways to compute it, e.g. using the following procedure:

1. A counter inside the `Update()` function is incremented by 1 at each frame.
2. When the frame rate is asked, the overall number of frames is divided for the number of seconds elapsed from the moment the test has been enabled

Given `FRAME_NO` the overall number of frames at time `TIMESPAN` in milliseconds, this metric can be obtained in this way:

$$FRAME\_RATE_i = \frac{FRAME\_NO_i}{TIMESPAN_i} \cdot 1000$$

A multiplier for 1000 has to be used since the $TIMESPAN$ value is expressed in milliseconds. This formula allows to obtain a version of the metric with a smooth trend.

It is not the only one method. The following formula gives a more precise idea of the variation of the frame rate in a small time window:

$$FRAME\_RATE_i = \frac{FRAME\_NO_i - FRAME\_NO_{i-1}}{TIMESPAN_i - TIMESPAN_{i-1}} \cdot 1000$$

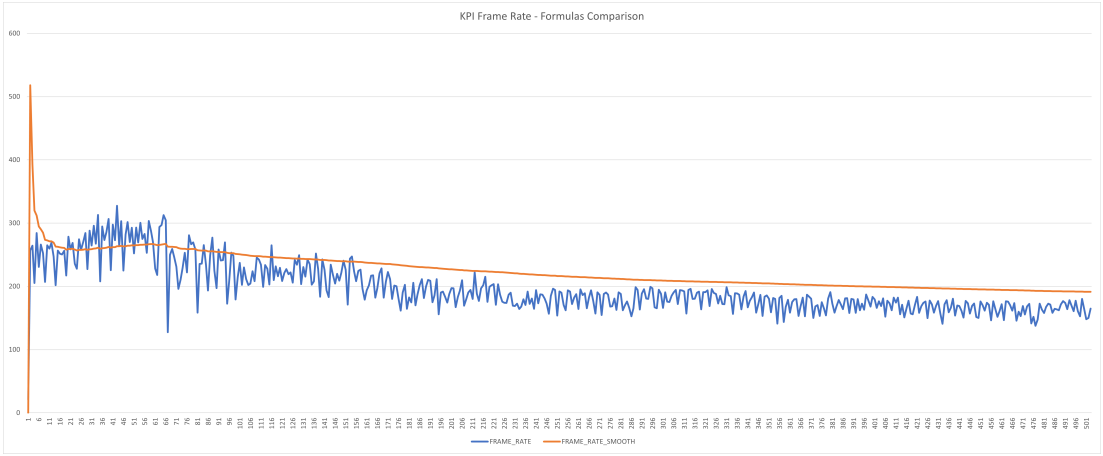
To give a idea of the different behaviours of this formula, figure 5.1 provides a visual comparison of the trends of the formulas. Despite the rapid behaviour of the second formula, the twos tend to have the same trend in time.

It is interesting to analyze the dynamic of the frame rate in conjunction with the other KPIs, and in particular the ones representing the load level of the system. So, let's discuss now the KPIs directly related to the positions database.

5.1.2 KPIs Hit Rate and Miss Rate

HIT rate and *MISS rate* are two KPIs useful mainly to understand if the localisation algorithm is working fine. This metric is tracked also in the chasing speed metrics report,

Figure 5.1: Comparison between two ways to compute the KPI frame rate; in orange the first formula, and in blue the second formula. Values are obtained from a simulation



related with all the other KPIs for positions database. Some interesting associations between this metric and the other ones will be introduced.

Here are the definitions:

- **HIT Rate**: the number of times the Positions Database recognized a already known position
- **MISS Rate**: the number of times the system detects the current position as new, and creates a new position

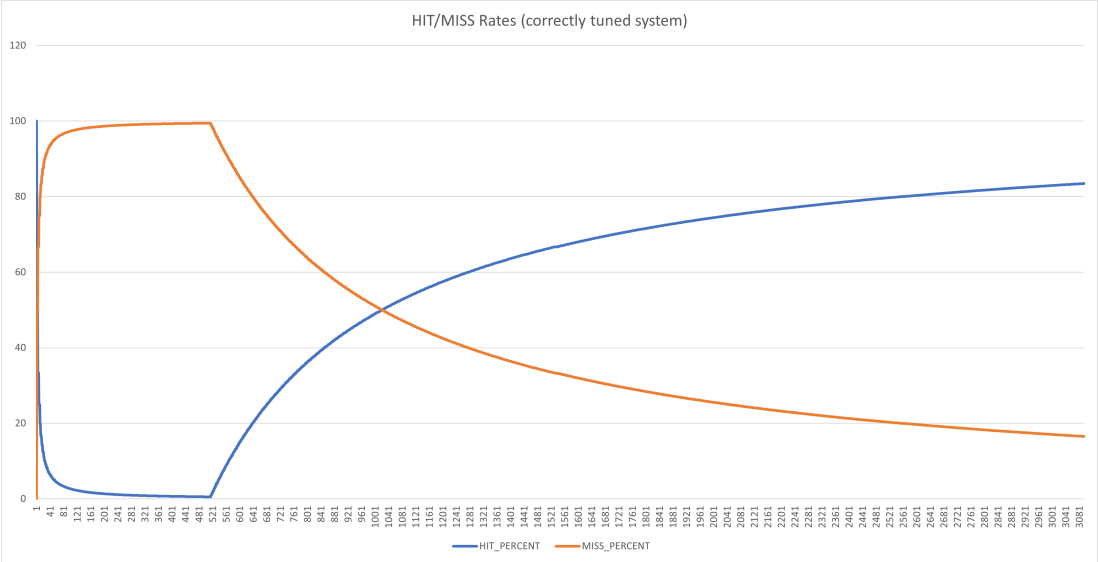
HIT rate and MISS rate are computed in percentage. Intuitively, a good tuning of the system will generate a low MISS rate, and the HIT rate will converge to 100% in the case the user performs a first exploration to remain then in the same area. A poorly configured system will produce more misses than needed, causing wrong position creations and anomalious configurations. Figure 5.2 represents the typical trend of this KPI: at first, MISS rate tends to increase due to the fact that the database is empty; as the system discovers new positions, the HIT rate increases.

A saturated MISS rate is not necessarily indicator of bad performances: it could happen that the user explores only unknown zones for a lot of time; but it is important that, when a already known zone is detected, the positions database is reactive enough to follow the user's position. To understand if the system is working badly, it is needed to compare this with the other KPIs.

5.1.3 KPI Sort Quality

HIT rate and MISS rate are good "external" values characterising the behaviour of the Positions Database; but, what about the minimum ordering time for instance? We need a KPI able to provide us informations about the "degreee of order" the list of waypoints

Figure 5.2: Typical trend of HIT/MISS rate, obtained in a situation where the system is working fine



has. For this reason, another KPI has been developed for checking the ordering of the list of positions.

This KPI comes directly from the design made for the dynamic sort algorithm. It can be observed that the algorithm tries, frame by frame, to locate the farest points at the end of the list, whereas the nearest one tends to go towards the head of the list. Taking the sum of the distances from the current point multiplied by their index, it can be observed that *the algorithm tries to maximize that sum.* This is almost the metric implemented in the chasing speed metrics report.

The basic idea considers the distances multiplied by the index where the point is at inside the list. Here's the formula:

$$Sq(P_{curr}) = \sum_{i=0}^{len(db)-1} i \cdot distance(P_i, P_{curr}) \tag{5.1}$$

The metric Sq reaches the maximum value when the array is completely ordered, whereas it reaches the minimum value in the worst case, i.e. when the array has the reversed order; in this way, it represents the internal dynamics of the algorithm.

If the system works fine, a convergence of the value Sq is expected, even if with some relevan fluctuations around the maximum value. On the contrary, when the system is not working fine, the metric shows a very slow convergence to the maximum value, signal of the fact that the positions database is not fast enough to "chase" the user's position. Figure 5.3 represents the behaviour of the sort quality when the system is working fine; figure 5.4 instead, shows the trend when the system is not working properly. Important to say that the important part of this metric is related to the

trend instead of the current value.

This metric, as formulated above, has a little mishap, related to the oscillatory behaviour around the maximum. In fact, when the user moves, all the distances are affected by this variation, leading to significant fluctuations to the disadvantage of the clarity.

As regularisation, a first idea is to modify the formula so that the distance is replaced by a multiple of the distance with respect to the base distance. This is the implemented approach as well, using a formula like this:

$$Sq(P_{curr}) = \sum_{i=0}^{len(db)-1} i \cdot floor\left(\frac{distance(P_i, P_{curr})}{base\_distance}\right) \tag{5.2}$$

This "trick" solves the problem partially. Here the base distance from the positions database is choosen as denominator in the relation, but it is also possible to take another value, however with the drawback that $floor()$ operator could make the metric "ambiguous". Let's consider the case, for instance, where all the distances are mapped into 0 or 1 due to a very large denominator.

This metric can be reviewed in many other ways. For instance, instead of the $floor$ function, a classification function can be introduces, mapping distances in a discrete set of points assigned with one rule; the basic idea used for the above-mentioned formula is almost this one, choosing a uniform and unlimited set of integers.

5.1.4 KPI Chasing Distance

Another simple metric can be implemented by evaluating the distance between the nearest point found by the system and the real position of the user from the internal odometry. This is the main idea of the so-called *Chasing Distance*, which can be found in the chasing speed metrics report.

The system continuously tries to find the nearest point to the user's position. Let's imagine the user starting from a known position, from distance zero. The user starts going forward, increasing its distance. At a certain moment, the distance will be slightly greater than the base distance, and in that moment the system will try to replace the identified position with another more suitable one. Let's suppose the position where the user is going is known: the distance, from a initial increasing, will start decreasing; the maximum in this case is approximately the base distance, but it could also be greater for a moment, depending on the efficiency that the application has in following the user.

And now, let's suppose that the next position is unknown. To ensure a good distribution of points in the space, the system will wait until the user is not approximately at a doubled distance from the previous point, to create a new one; in that moment, the distance will decrease quickly, but before that, the distance will reach approximately the double of the base distance.

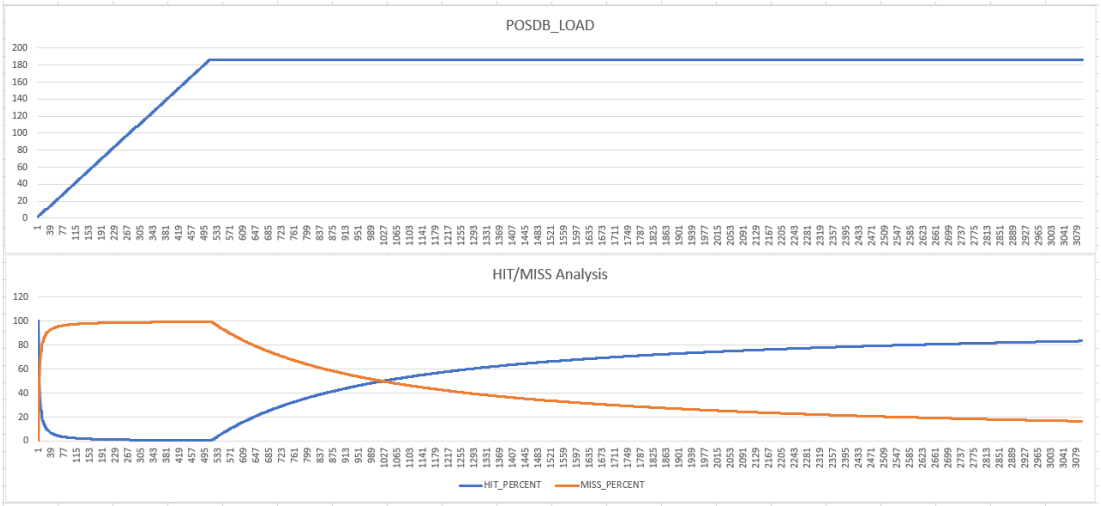
Figure 5.5 shows clearly how this KPI works when the system is correctly tuned, compared with HIT/MISS ratio. Please notice that there are some cases in which the

metric is below the base distance for long time without a "spike", meaning that the user is "going around" in the same waypoint. Values of the spikes are not precisely the base distance, since there's also a tolerance among the parameters: a new position is discovered when the position falls in a band "around" the frontier, and not precisely in the frontier.

5.1.5 KPI PosDB Load

The *PosDB Load* indicates how much points are inside the positions list: it is a simple items counting. It grows then a MISS happens, and it keeps the same value when the user is still, or moving among knonw positions only with a good behaviour of th esystem. It has a monotonically growing trend. Its behaviour is clearly shown in figure 5.6.

Figure 5.6: Hit/Miss Vs. PosDB Load in straight line benchmark.



This simple KPI allows to evaluate the impact of the memory allocation to the other KPIs describing the behaviour of the system. In particular, it is interesting to combine this metric not only with the frame rate, but also with the average number of swaps per call and the busy time.

5.1.6 KPI busy time

This metric is about not the dynamics of the system, but more on the resources requirement. The KPI busy time is defined as the average time spent by each call of the function `SortStep()`: in simple terms, the computation time per frame. The way it is derived is simple: it is the difference between the time the call started and the time the call returned. In the Chasing Speed Metrics report, this metric is computed in two ways: in average, and in terms of maximum time. Here's the precise way used for computing this metric:

- A counter of calls is kept during the execution.
- The time spent for one execution (in milliseconds) is summed to the time spent by the other ones.
- When required by the analysis script, the value is computed dividing the time spent by the number of calls, obtaining a average busy time per call.

Notice that, as said before for the framerate, this metric gives a overall idea of the trend, but tends also to become "lazy" as the working time of the system increases.

This can be related mainly to the number of clusters, besides the time spent by the algorithm to coordinate itself. Also the maximum frame rate of the device is a parameter affecting how this metric behaves: if the device is "slow", the average busy time increases.

5.1.7 KPI Swaps per call

This metric offers a perspective between the busy time and the sort quality, showing the load per frame and also considering the ordering of the list. Ad basic idea, given a certain number of active clusters, in the worst case the algorithm will perform a number of swaps leading towards the number of active clusters. This KPI computes the average number of swaps per call since the system started to work.

It is computed with the same procedure described for the busy time, but using the sum of all the times the `swap()` function is called as numerator. The complete ordering is reached when the number of swaps decreases to zero: this means also that the user is still in the same point, so the metric will tend to zero. In the worst case, the metric tends to its maximum, that is the number of active clusters, meaning also a overload of the system. If the system is working properly.

5.1.8 KPIs Request KB and Response KB

Let's start talking about the metrics regarding the server integration from the client point of view. The first one is the weight of the requests and the responses.

- **Request KB** is the weight of the requests in Kilobytes
- **Response KB** is the weight of the responses from the server in Kilobytes

The sum of the two metrics is the entire amount of data exchanged between the client and the server. With the structure choosen for the application, it is quite easy to extract them: count is done before starting the HTTP request and after the HTTP request returned, in only one point of the entire application.

5.1.9 KPIs Latency and Minimum Bandwidth

This metric is the time spent by the call from the start of the request to the moment the request returns. It is computed in the same way of the busy time. Since it is not possible to separate the net processing time by the server from the overall time, this time can't be used for the computation of the bandwidth. Bandwidth can be approximated with the ratio between the overall kilobytes and the latency: this value will be less than the real bandwidth. This way of reasoning can be used to compute a *minimum bandwidth*.

5.2 Application Testing

Most of the tests aim at assessing the performances of the client application, mostly in simulated environment, but also with practical tests on the device itself, connected with the remote server.

Let's start discussing the methods implemented for practical tests; after that, practical experiences are showed and results. First part is dedicated to simulated tests. Later on, device tests results are introduced.

5.2.1 Benchmarking, Reports and Logging

Figure 7.9 provides a complete overview of the app architecture, showing also the components for testing and data extraction in light blue for functional components, and in green for data streamings.

PlayerBehaviour represents a block implementing one benchmark test. It is a sort of test, expecially meant to be used in simulation, moving a "player" (i.e. a Unity GameObject) in the area. Benchmarks are useful to inspect the correctness of application logics and to estimate overall performances of the application in simulated environment. Obviously, user tests on the device are needed to assess the user experience and to be sure that the tuning works fine, but simulations can give a good former idea of how the application works under certain conditions.

Throught benchmarks, the system tracks the movements of one "Player" GameObject which can move in a particular way, both randomly or in a predefined shape, with some parameter, e.g. the movement speed. Three important tests have been created for this project:

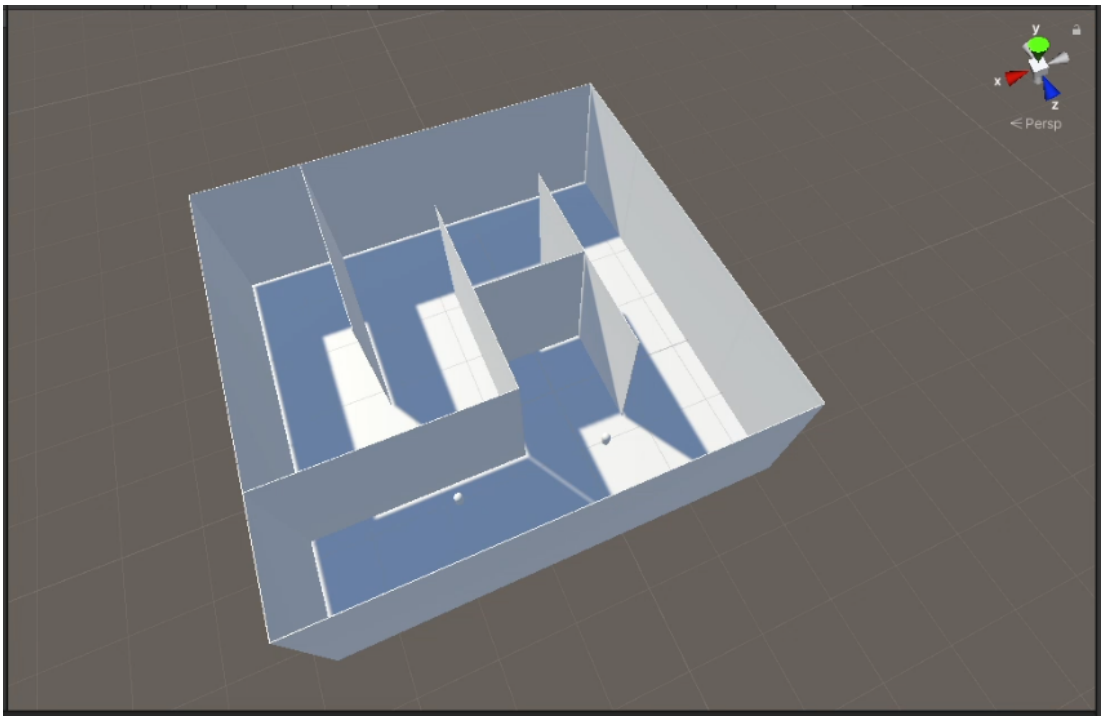
- **Straight Line Benchmark** : the player moves along a simple straight line with a given length at a certain velocity, up and down.
- **Waypoints Benchmark** : the player moves on a custom map, made of landmarks. The previous test can be seen as a particular case of this benchmark, using a path with onyl two landmarks. It is possible to make the player move either randomly in the path, or enforcing the passage thought some particular paths.

The first benchmark allowed to obtain interesting results about the application, even if in simulated environment. The second one has been used especially with random paths, since it can simulate the user's activity better than a well defined path. All the results are collected in the reports, analyzed at the end of the experiements.

5.2.2 Simulated Experiments

First tests performed are simple performance tests about the performances of the visuals in simulated environment. Objective of the test is to detect possible losses of frame rates due to the computational requirement of the visuals. Tests have been performed in simulation evironment at first, to confirm with a quick user test on the device.

Figure 5.7: Test Map, used for development and testing in simulated environment.



5.2.2.1 Visuals Stability Testing

The user moves freely (no benchmark) inside a test map (figure 5.7). Two visuals are tested at time: the surroundings visual, and the minimap visual. Here's the procedure followed to perform simulated test:

1. After enabled the simulation, visual is turned on; the user explores the entire map, and then stops in in one point for a while.

2. Visual is turned off and the user keeps the same position for a while.

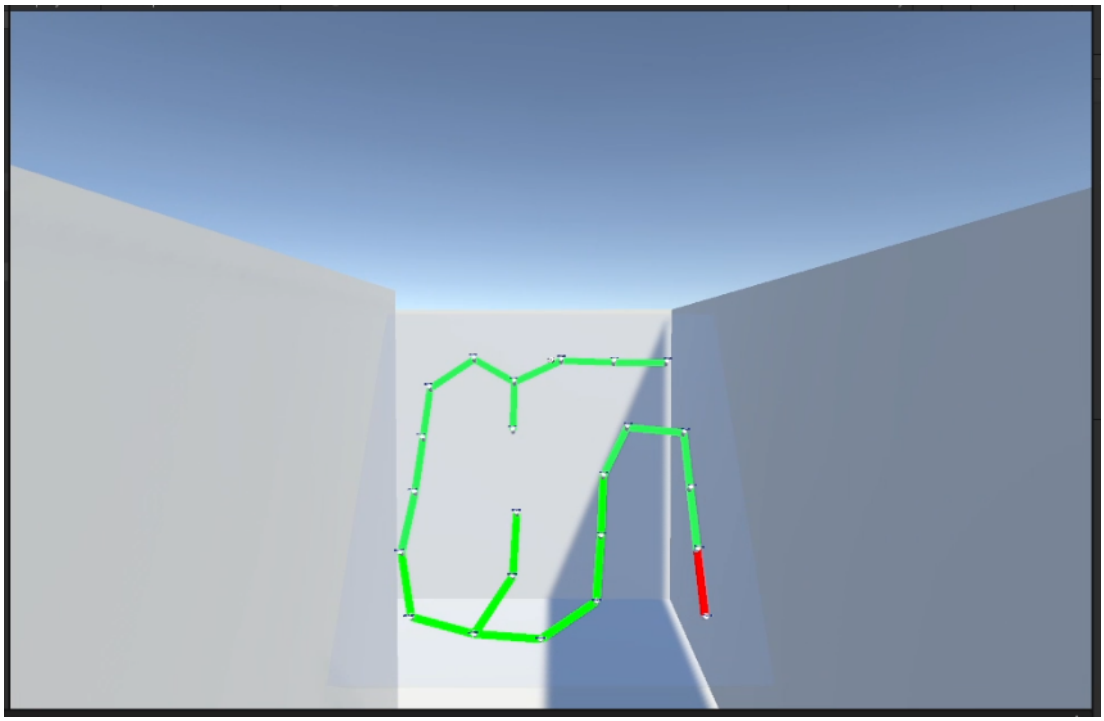
3. Visual is turned on again, and the user starts again to go around; in this case, all the points of the map are known.
4. Visual is turned off again and the user keeps the same position for a while.

Positions database have been tuned usign a "minimal" setting (please refer to the other tests); here are the values:

- Cluster size: 15 items per cluster
- Max Indexes: 15 indexes
- Base Distance : 0.5 meters, Base Height : 0.8 meters
- Tolerance : 0.05 meters

Of course, this is a minimal test: using a simulation environment, performances are naturally higher than using the device due to the performances of the PC; this test is useful only to check if there are important lags that could compromise the overall user experience. To have a better estimation even in this case, a minimal configuration for the positions database have been chosen (i.e. with a lower base distance, the system could start generating anomalies in positions tracking, as explained later). Since visuals are updated only when a change of position is detected, frame rate could be stable when the user is still, changing framerate only when the user moves. In the following, the main data resource is the Chasing Speed Metrics report.

Figure 5.8: Visual Minimap during the visuals test, after explored the entire area

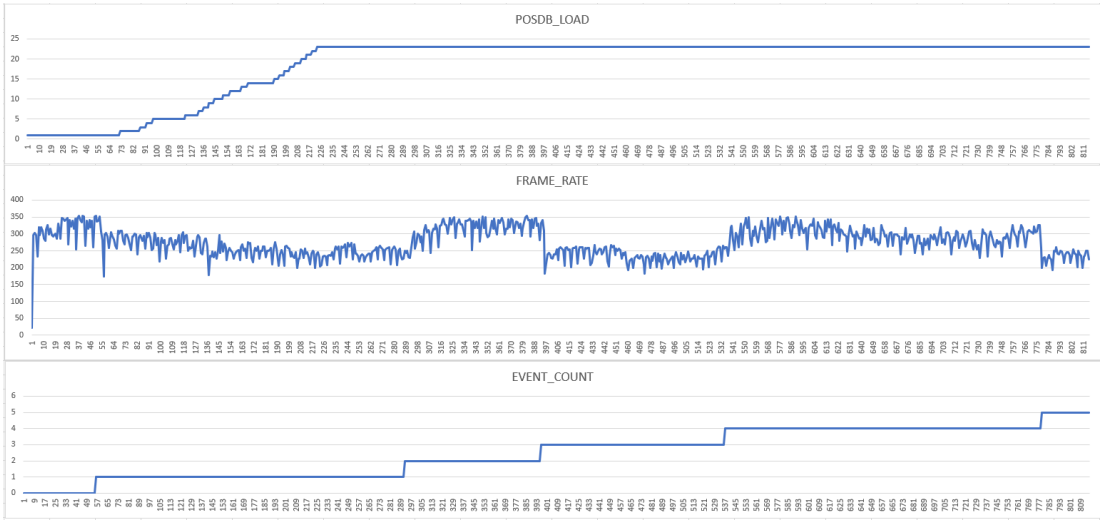


Minimap Visual Testing. A simulated test have been performed with the procedure detailed above. The report has a field, `EVENT_COUNT`, increasing of 1 when a change in visuals is detected: this allows to divide the report in parts, as well as to detect when the user changes visual. This is important to split the frame rate based on the events.

Simulated environment can provide a frame rate mostly between 200Hz and 350Hz, very high compared with the one of the device which has a maximum of 60Hz when no recordings are ongoing.

To have a complete analysis, at least 3 quantities are needed in this case: from the top, `POSDB_LOAD`, `FRAME_RATE` and `EVENT_COUNT`. Figure 5.9 shows this analysis.

Figure 5.9: Visual Minimap during the visuals test, after explored the entire area



The analysis 5.9 shows a gradual decreasing of the framerate with enabled visual as the player collects new points. 23 points have been collected during this first exploration; after this phase, the user don't discover new points.
The central part is interesting: when the visual is turned off for the first time, the frame rate increases, a bit slowly since the user moved for one moment after the change of visual. Next, with still user, visual is enabled again, and it produces a rapid decreasing of framerate. During the final exploration with not enabled visual, the curve tends to decrease since the positions database works in background for continuously chasing the position of the user.
Even if it is a small test, considered the number of points, it is useful to understand how the performances change, and this behaviour is acceptable: the framerate remains bounded in the same interval in runtime with a certain regularity. It is also possible to estimate the forecasted framerate on the device, using this simple procedure:

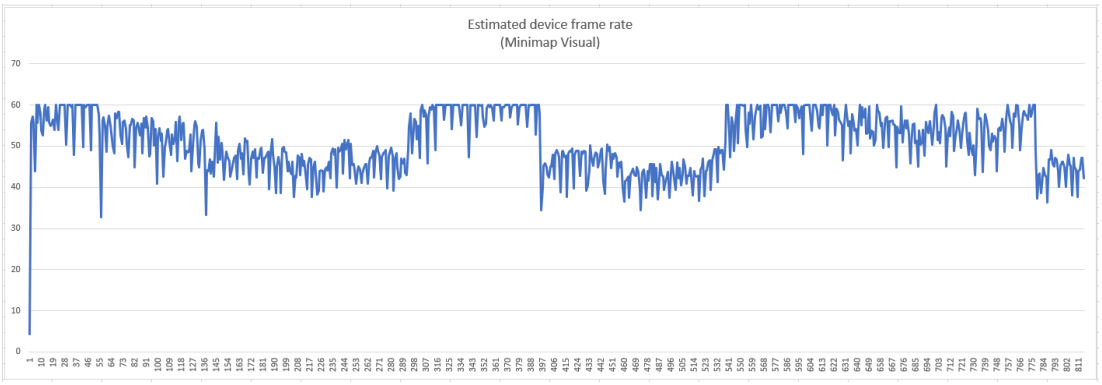
- (Max Reference Frame Rate) Compute the maximum of the frame rate (I suggest a percentage of the maximum for a more realistic test, for instance 90% of the

maximum);

- (Percent Frame Rate) Compute the percentage between the measured frame rate and the maximum frame rate;
- (Estimated Device Frame Rate) Using 60Hz as device maximum (confirmed by experiemnts), take the minimum between $deviceMax * percentFrameRate$

This simple computation allows to estimate the device frame rate; most of the times, it represents a worst-case scenario, compared to the real measurements. This estimation is absolutely important to evaluate the user experience, especially in Search and Rescue: a laggy or glitchy visual could distract the user. Figure 5.10 shows the result of the estimation: minimum is over 30Hz, which is good.

Figure 5.10: Estimation of device frame rate given the one from the simulation using minimap visual.



Surrounding Visual Testing. Testing proceds in a way almost identical to the one described before. For surrounding visual, it has been used a depth of 10 meters from the current position. Positions database has the same tuning as before. It can be observed a identical behaviour as before: the trend of the measurements is almost the same.

5.2.2.2 Positions Database Tuning

This solution requires a tuning process to find the settings allowing the localisation system to perform as efficiency as possible. So far, this process has to be manually performed with simulated tests, mainly based on HIT/MISS rate. When the system is correctly tuned, the positions database is able to create new positions only when needed, as well as to recognize already registered positions; in bad configuration instead, the localisation system tends to create positions even if the user is located at a known position. This anomalous situation can be seen using one of the visuals implemented in this project, as well as detecting sone "symptoms" from reports.

All the tests are performed varying the four main parameters of the localisation system, in the test map with users movements decided manually. Parameters are: base distance (not the base height), tolerance, cluster size and max indices. Also the movement velocity has to be considered and changed during the tests.

There are some tests particularly important for evaluating and stressing the system, as listed here:

- *Drift Test* : user moves in the test map with the highest velocity as possible, preferring continuous movements.
- *Stop and Go* : the user remains in the same place, waiting; after a while, the user starts walking again quickly.

In all the tests, a configuration is set before starting with the test, and then the test started following a procedure like this:

1. Slow exploration of the entire environment
2. Slow movement among the known points, to test if the positions database is working fine at slow speed
3. The user keeps the same position, waiting the positions database to reach a complete ordering of positions
4. The user starts then with quick movements in the area, stressing the system among the known points.

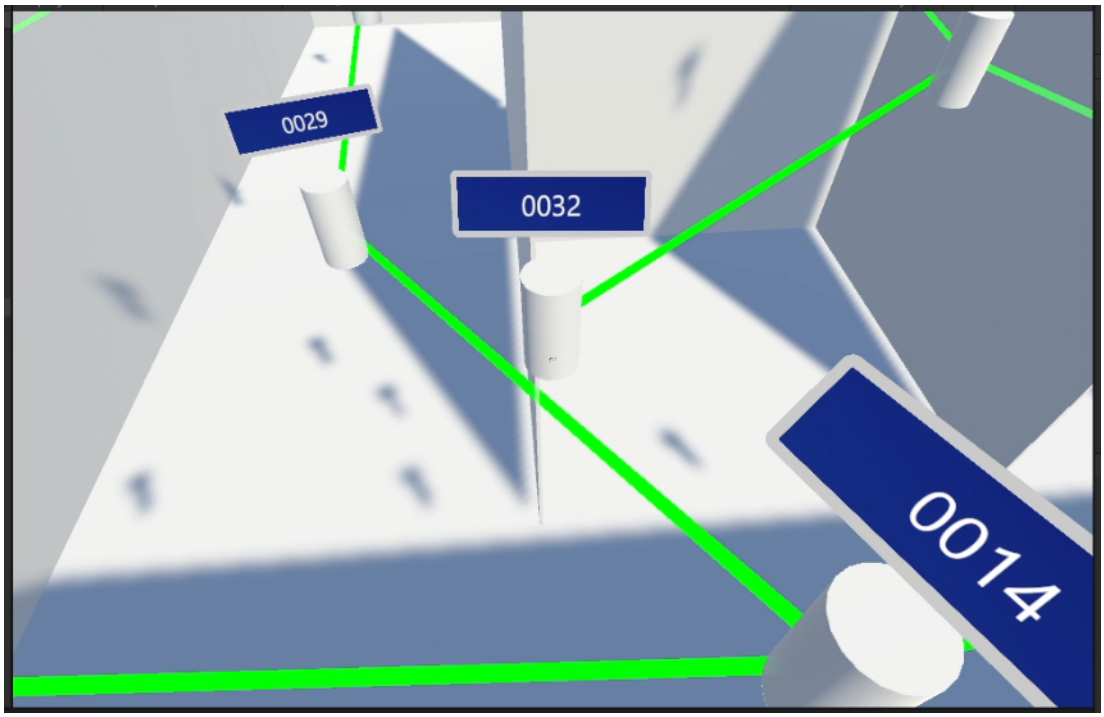
Unity moves the main camera at approximately 1 meter per second in straight-line movement, and 1.4 meters per second in quick drifts; a slightly brisk walk has a velocity of 4.5Km/h, that is about 1.1 meters per second.

Minimal Conditions without Optimisations. Let's start from a situation where only base distance and tolerance are considered, turning off the cluster optimisation and the max index. Using such a setup, the objective is to find out the minimum values that allow the system to work fine. Tests are performed in simulated environment.

Here are the main results, briefly summarized.

- **Base Distance: 1.00m, Tolerance: 0.15m**: considering a realistic application, one meter is a reasonable distance. Generated 16 points (no anomalous miss); perfomances are quite good, even if the path tends to simplify too much the shape of the path (it is mainly due to the test path in this case, which hace too much narrow passages).
- **Base Distance: 0.50m, Tolerance: 0.15m**: the system still tends to have good performances also in this case. 35 points have been generated, of which 31 from the base map. Some of them have been generated during drifts, but the movement have been really quick. Figure 5.11 shows the situation. Movements not too much slow don't generate misses.

Figure 5.11: A possible anomaly, generate in test number 2 of the minimal tuning assessment. Due to the high velocity of the user, one waypoint have been generated "in the middle of" other two waypoints.



- **Base Distance: 0.25m, Tolerance: 0.15m**: It has been said that this base distance is not realistic, thinking in practical terms; it is a stress test. 42 points have been generated, but in this case, a great numbe rof anomalous waypoints have been inserted, creating a total number of 133 points. Figure 5.12 represents the situation: the positions database is no more able to correctly track the user's position. Almost all the anomalous positions have been generated after a simple start and stop: using a configuration like this requires that the rescuer has to move with caution.

It seems that, with no optimisations, the minimal configuration stands in a value between 0.50m and 0.25m; the issue us due to the fact that only one idex has to scan the entire list, at a given frame rate. As the list becomes longer and longer, the risk sould be that the system finds itself involved in reordering values which are too much far from the current position, so when the user starts moving, the first part of the list is not yet ordered since the index is somewhere else, leading to delays in following the positions, and then to a disaster in terms of system performances.

We could choose a very large base distance, especially for example when the user is exploring large outdoor environments: in these cases in fact, it is not required a extreme precision in paths tracking. But, in indoor situations, we need more precision; one of the anomalous situations in this case is for instance the possibility to generate

a link traversing walls, i.e. a wrong path. So, a tradeoff is required: decreasing the base distance improves the precision of the path, but also creates more waypoints to manage. 0.50 meters is works good, but anomalies are still possible, and let's consider that the test considers only a small environment: what could it happen with larger environments? So, it is not enough to adjust the base distance: we need to use a optimisation.

By further tests, it can be found that the minimun base distance without optimisations is about 0.50 meters with a very good precision; every value lower than this generates the wrong behaviour found in the third test.

Minimun conditions with Cluster Optimisation and Max Indices. As explained in Chapter 3, cluster optimisation splits the array into *cluster*, and handles a separated index for each cluster, better distributing the update along the entire list.

What has been done here is to search for the minimum base distance given a reasonable cluster value; tuning worked on two parameters this time. Here are the tests performed in this case:

- **Cluster: 4, Base Distance: 0.45m**: 29 points generated in the first part of the exploration, 33 points at the enf of the exploration, 8 clusters opened; no misses during the inspection of already known set of positions; the four added points are due to the drift movement, of which only one is a real anomaly. It is a progress: without optimisation, the same situation produced anomalies.

- **Cluster: 4, Base Distance: 0.35m**: 34 points in the first exploration (with 2 overlapping anomalies), 39 final points (mainly due to not precise drift movement, only 2 anomalies), 9 final clusters. This configuration works quite fine.

Using this optimisation, we managed to decrease the base distance of 15 centimeters until 0.35m instead of the previous minimal configuration. This is good.

But, what about performances? And in particular, what about the framerate of the system as the number of clusters increase? In the latter test, we opened 9 clusters, so we should expect 9 checks in the same frame. We should expect that, as the list increases more and more, the number of indexes per frame will become so high to produce a relevant aggravation of the system performances, clearly visible in terms of frame rate.

To limit the loss of performances, we could act in three ways: incresing the number of clusters (4 clusters is just a test, but it is a really small value); putting a maximum number of updates per frame, which is the max index optimisation; and increasing the base distance, even playing with the tolerance. With these parameters, it is possible to find a good tradeoff which enables the system to work.

5.2.2.3 Straight Line Benchmark Tests

Another way to stody the behaviour of the system, and to find a correct tuning of the system, is to use one of the two benchmarks implemented for this project. In this case, the nechmark moving the operator along a straight line, forward and backward. The

main advantage of this kind of test is the possibility to fix also a precise velocity as test parameter, as well as to regulate the distance of the path, thing almost impossible in the previously tests for database tuning.
Parameters of this test are the folloing ones:

- Base Distance: 1m for all the tests
- Tolerance: 0.2m for all the tests
- Cluster size: 10 for all the tests
- Max Indexes: 10 for all the tests
- Operator's velocity (*variable*)
- Path Distance (*variable*)

A set of 8 tests have been performed. Here's a summary:

- **Velocity: 0.64m/s, Path Length: 10m**: 8 points generated. Absolutely stable.
- **Velocity: 1.46m/s, Path Length: 50m**: 38 points generated, still stable. The velocity is 5.25Km/h, which is a quick walk.
- **Velocity: 3.62m/s, Path Length: 50m**: 37 points generated, still stable. This is clearly a unrealistic velocity: just for stressing the system.
- **Velocity: 6.0m/s, Path Length: 100m**: 74 points generated, still stable despite the velocity.
- **Velocity: 2.75m/s, Path Length: 250m**: 313 points generated with a large number of misses, **Not Stable**. In this case, the path is long and the speed too much. Notice that this velocity is about 10Km/h, which is a slightly sustained running, not realistic.
- **Velocity: 1.67m/s, Path Length: 250m**: 186 points generated, **Stable**. This is a important result. The velocity is about 6Km/h: the user is about to run.

So, looking at the results of the simulations, the system works fine when the user is not running, which is the typical case. Obviously, these tests heavily depend on the performances of the PC: more tests on HoloLens2 have to be done in this way, even if without benchmark (unless a system is found to estimate real performance based on simulated performance).

5.2.2.4 Waypoints Benchmark Test and Server Test

This final test involves the entire architecture. In this test, the player moves randomly from one waypoint to another one. A square shape has been built. Here are the settings used for the test:

- Positions Database (fixed)
  - Base Distance : 1.5
  - Base Height : 0.8
  - Tolerance : 0.2
  - Cluster Size : 10
  - Max Indexes : 10
- Client Utility (changed during tests)
  - Upload Time : 10
  - Download Time : 21
  - Update Radius : 999
- Using stub calibration

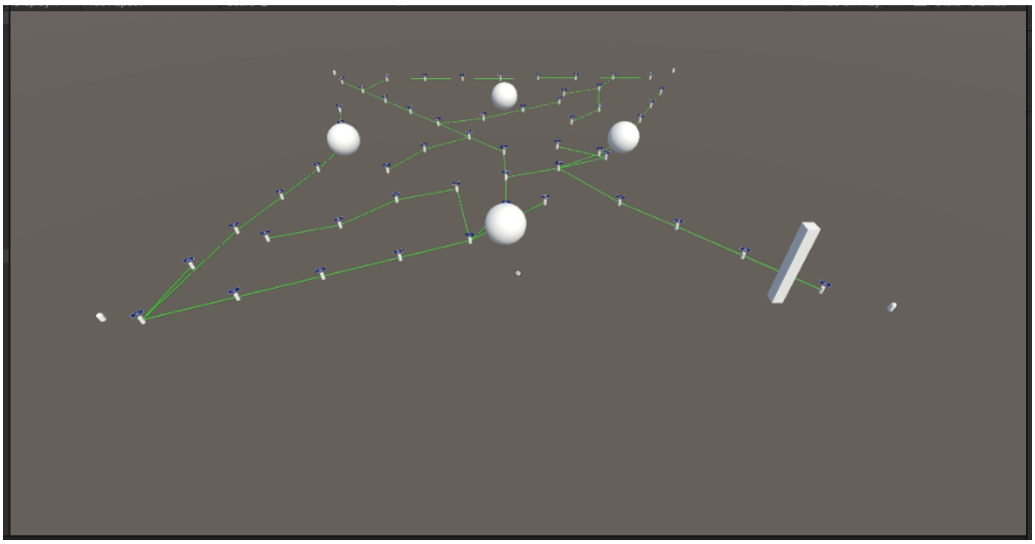
As described in the chapter 4, the server side applied different effects on the data: we may expect that data are integrated with the ones already known by the device, even during exploration. During the exploration, the device uploads new data every 10 seconds, and the server tries to align the two dataset each time. Uploaded data are also filtered, so that when data are downloaded from the server, they are cleaned by possible anomalies previously loaded into the database.
Here is a summary of the tests performed in simulated environment.

First Exploration from Scratch. The player moves around to create a first map. This map should not be too much complete, in order to be able to test also the integration of new data.

Second Data Integration. At first, the system downloads the data from the server: the known path at the beginning of the first test is perfectly equal to the one at the end of the previous experiemnt. New points are added.

Third Data Integration. new parameters: Upload time 13s, Download time 10s, update radius 3.75m. Since the reduced update radius, less data are returned from the first download. Subsequent exploration passes on data already discovered by the client in previous sessions and unknown in this session, making the server to re-align the positions from the client with the ones already known. While the user goes around, paths around are integrated automatically from download requests. Sometimes the download process generates occasional anomalies due to the quick continuous movement of the player (velocity around 1.5m/s). Figure 5.14 represents the final situation.

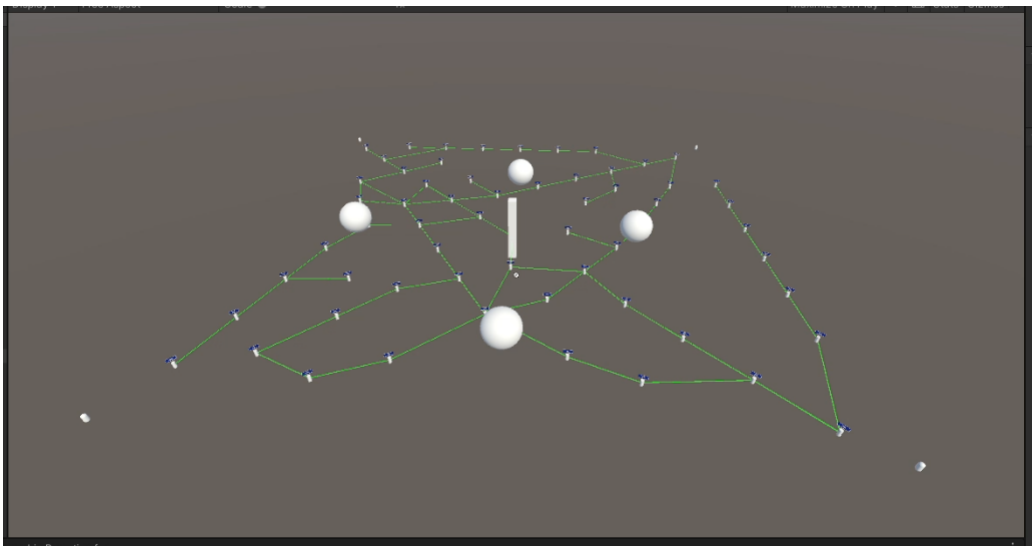
Figure 5.14: Server simulated test, third session: some anomalies are generated by the system during the download, due to the quick movements of the player.



Fourth Session. Using the same parameters as before. New paths have been discovered during this test, showing that the server is capable of mixing different sessions.

Fifth Session. Update Radius is changed to 999m (as the largest as possible) in order to force the system to download all the points, and to see the final map then. Result is interesting, since it contains also paths found in other sessions (figure 5.15)

Figure 5.15: The final map as result of different sessions mixed in one dataset.



Anomalies from the previous steps disappeared in the final "mixed" data structure, since the server is able to detect anomalous points and to filter them out. The structure returned by the server contains cycles since the server returns all the passages returned by all the operators in all the sessions. Noticeable that there's no superposition of archs, another possible anomaly discarded by the server application.

5.2.3 Device Experiments

Tests on the devices are performed in terms of simple User Acceptance Testing, i.e. with the objective of assess the stability and usability of the application in practise. Clearly, more experiments are required to understand all the details of the application in terms of the above-mentioned KPIs.

Let's start discussing the setup of the application, to discuss in the end the results of the tests.

5.2.3.1 Application Experimental Setup

A first modification of the application has been the implementation of a small hand-menu collecting all the main settings of the application. In fact, voice commands are not efficient at all: the user has to repeat the command many times until it is not understood by the system, and the processing of the voice command only requires a number of seconds, fact that reduces the usability, especially with respect to the needs of a Search and Rescue operation. Figure 5.16 shows the panel implemented, shown only with clearly opened palm in front of the camera in order to avoid situations in which the panel appears without a clear intention by the user.

Figure 5.16: The app Hands Menu. The leftmost column contains the three visuals. The right down button makes the leftmost info panel appear.

The HoloLens2 employs the explicit calibration: it is necessary to be able to communicate with the server as well as to correctly integrate data into the server.

Figure 5.16 also shows the configuration settings of the app. It has been decided to use the same configuration of the final server test.

The final application uses onyl two reports: Chasing Speed Metric and Client Stats, plus the logging, set with the highest level of detail for testing purposes.

5.2.3.2 Application Tests

Tests performed in this case are referred to simple usage of the device. After calibrated the device, the user goes around exploring the area, and trying all the visuals provided, with the above mentioned configuration. In particular, the experiement has been divided in two sessions: the first one with the objective to collect the first data, and the second one which downloads part of the data from the previous session and integrates other data.

A first important difference, as said, can be found in the frame rate. HoloLens2 enforces the frame rate to be 60Hz at most. During recordings, HoloLens2 forces the frame rate to only 30Hz, which creates some complications in interacting with the user interface.

Estimation of tuning was found to be correct from the experiments: the system works fine, especially in the first experiment; the user's velocity reaches 1.4m/s seldom, preferring a maximum of 1.0m/s with some stops.

First Session. The first step is the system calibration: the user selects the option in the hands menu, and follows the procedure. When the calibration ends, the user can go around, using the visuals to explore and inspect the path.

At the beginning of the test, the database is completely empty: they are uploaded progressively during the exploration. Figure 5.17 gives a idea of the entire path explored, and how the user sees the path in the visuals from two perspectives (surroundings visual with maximum depth).

During the test, framerate is very low (30Hz at most), but stable and without sudden downspikes. 21 points have been generated with a average velocity of 0.6m/s. No anomalies have been detected during this phase.

A part of the environment has not been explored during the first session, in order to integrate later more data.

Second Session. As first step, the system downloads 17 points, not 21 since the update radius does not cover all the area explored before; these data are downloaded later, when the user is near enough to those points. Sometimes the system generates anomalies due to the large number of waypoints integrated in a moment into the database: the system requires a bit for preparing data after the download.

Another test with low frame rate due to the recording, but still stable and uniform. 3 anomalies have been generated, despite the correct tuning of the database; at the end of the test, 33 points were in the database, of which 17 previously downloaded (and some of them not visited during the test). Maximum velocity was 1.0m/s with a number of stops.

Figure 5.3: Trend of the KPI Sort Quality when the system is working properly. User is moving up and down across a straight line. Behaviour of the metric tends to be bounded after a while.

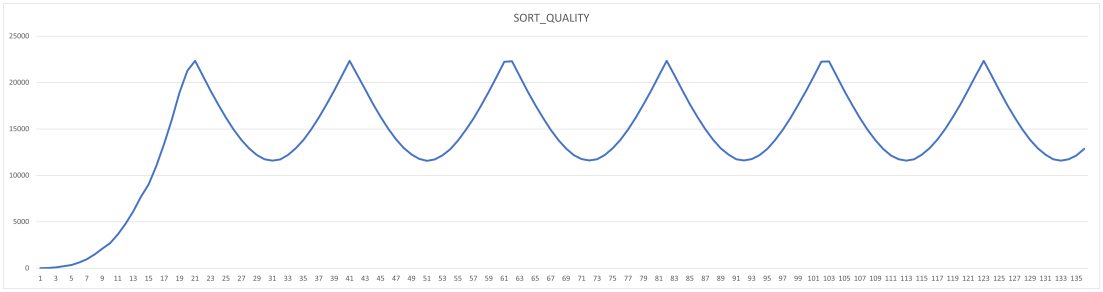


Figure 5.4: Trend of the KPI Sort Quality when the system is not working properly: in this case, the metric tends to diverge due to the continuous increasing of the number of points; the system is not able to properly follow the user's position.

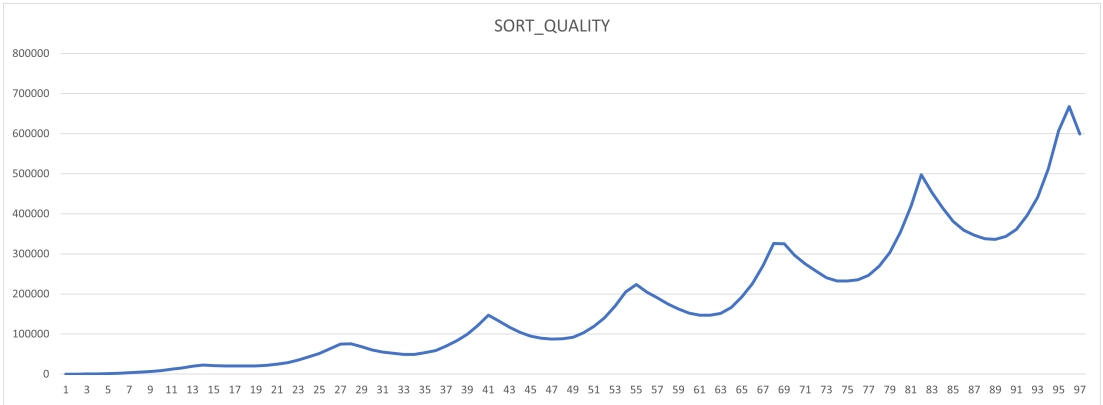


Figure 5.5: The following diagram shows the comparison between the HIT/MISS trend and the Chasing Distance. The user is moving along a straight line in simualted environment.

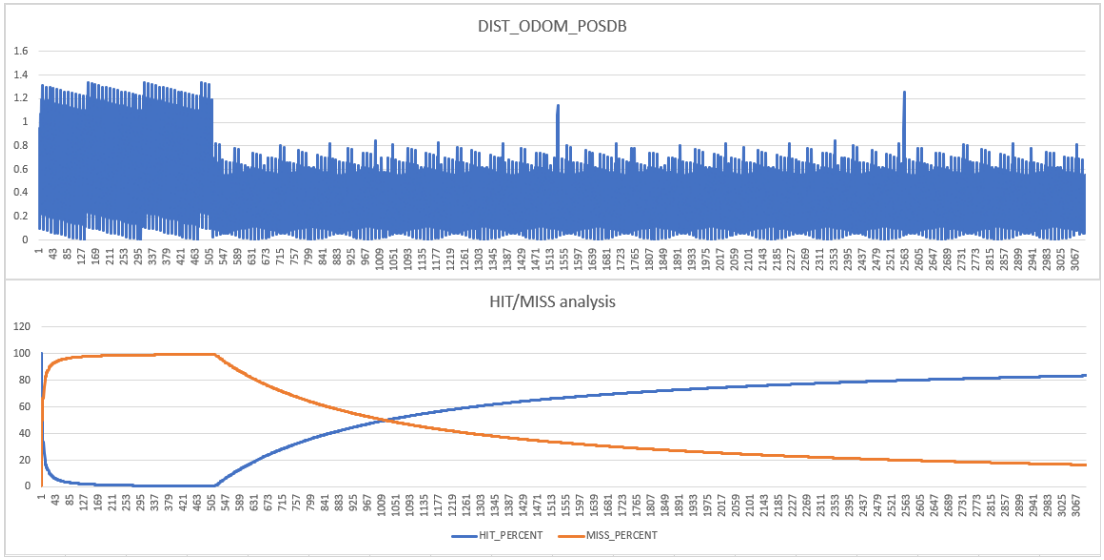


Figure 5.12: Base distance 0.25m with no optimisations: the database is no more able to correctly follow the user's position, causing the generation of a enormous number of useless points.

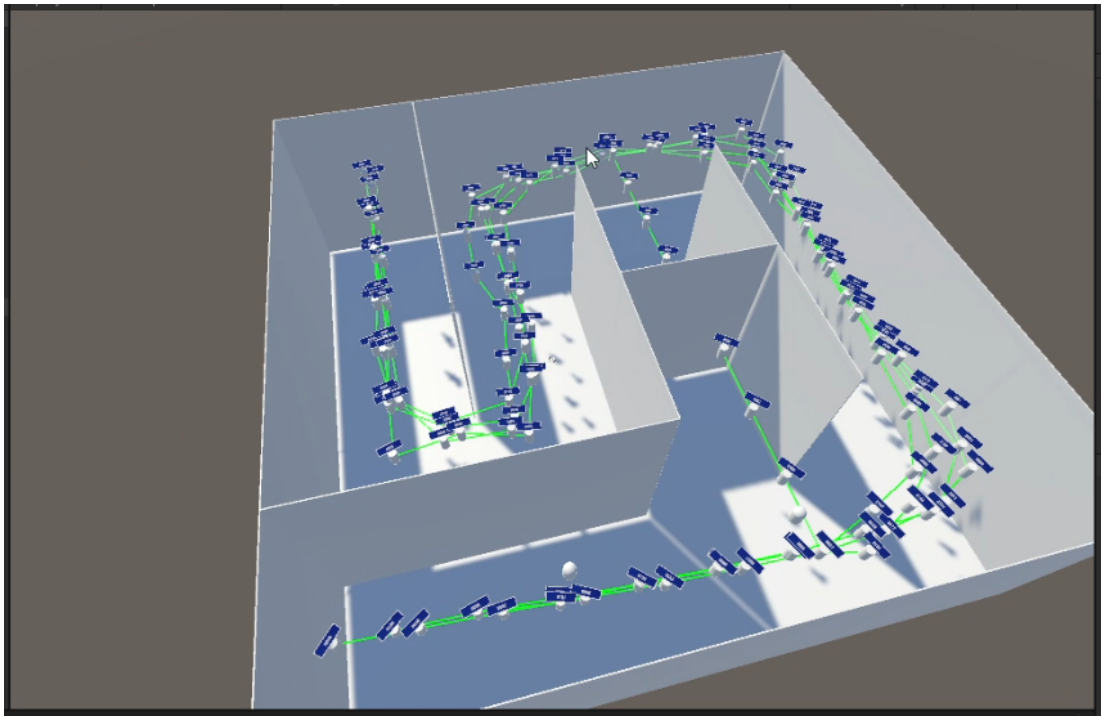


Figure 5.13: Testing Environment for the Positions Database Tuning; it is pointed out the Unity component providing a live estimation of HIT/MISS ratio.

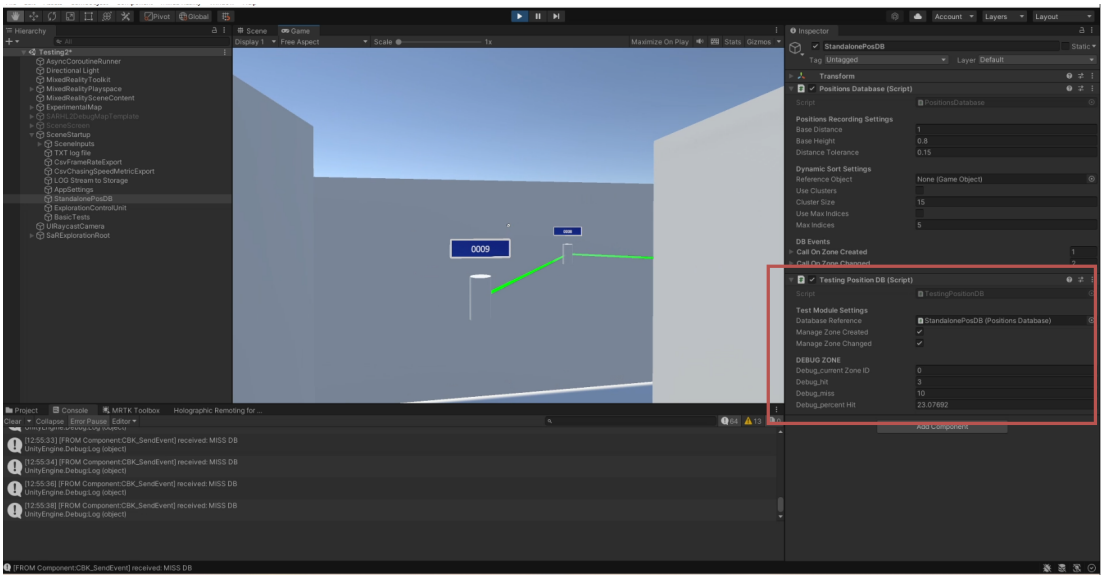
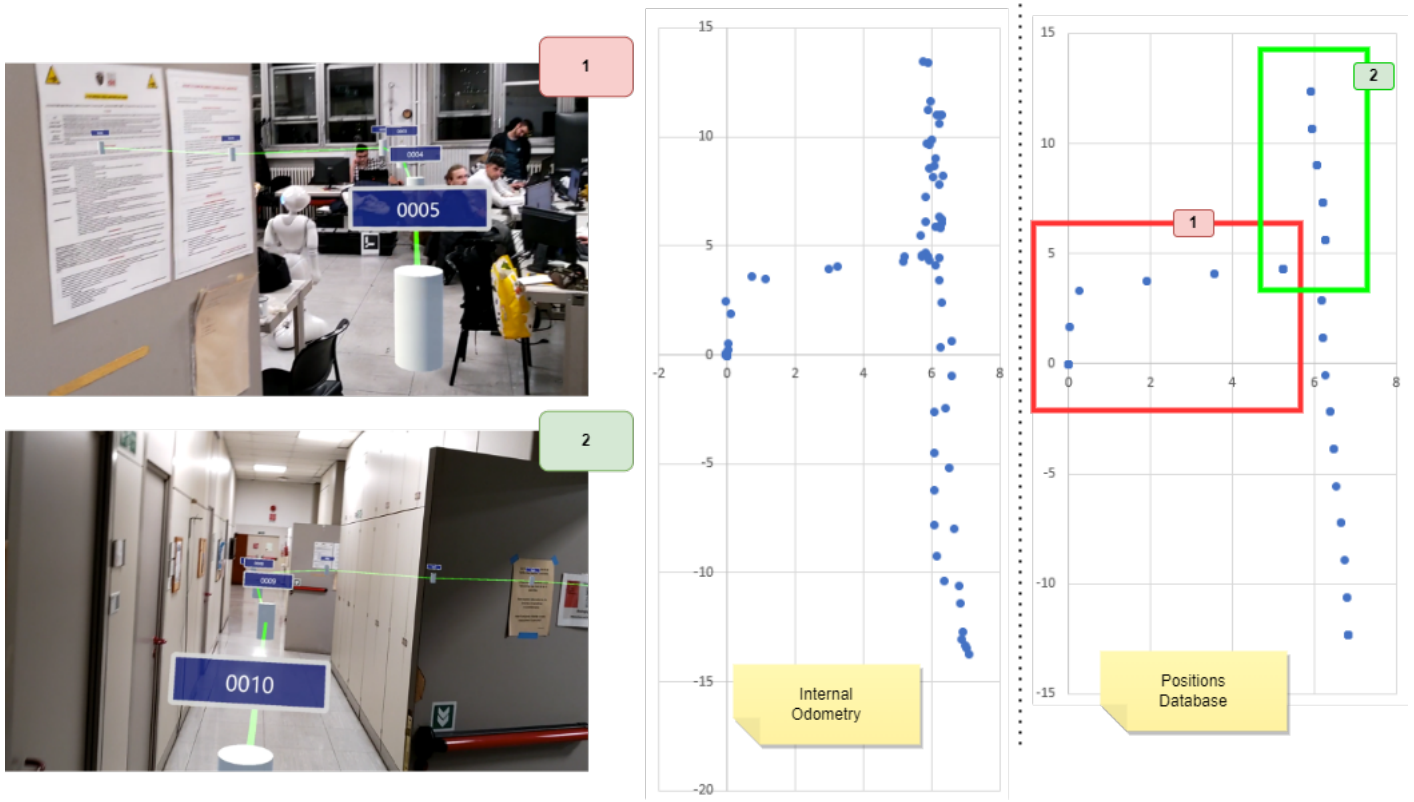


Figure 5.17: Path of the first test on the device, with user's perspective, values from internal odometry, and sampling from positions database.



Chapter 6

Conclusions

6.1 Results Recap

Below the main steps and results of this work are briefly summarized:

- The idea for the proposed applications started both from the previously implemented project (Testa [2022]) and from a paper review in search of all the main operative aspects and environmental challenges involved in Search and Rescue operations. A main idea of the architecture have been developed after this phase.
- The first stone of the project was the design of a data model able to track the user's activity in efficient way, and ligh enough to be shared among other devices with no preprocessing.
- Beginning with the implementation of the HoloLens2 part, a framework for efficient visualisation has been developed; next, three visualisations have been proposed: Path visual (tracking path as the user walks around), Surroundings visual (each time the user moves from one waypoint to another one, the system finds all the paths around) and Minimap visual (a minimap following the user and showing all the paths around). A model-view-controller model has been applied.
- A significant part of the work on the client has been done on the localisation and mapping system: following the data model designed at the beginning of the work, it has been developed a localisation system based on sorting a list frame by frame, highlighting the assumption that, within a common walk, there are not so much items changing positions from one frame to another. After that, a mapping system capable of creating new waypoints have been implemented. Optimisations have been required in order to achieve better performances; two in particular have been developed: cluster size, and maximum number of indices.
- To make positions shareable, it is needed to have the capability of tranforming local HoloLens2 coordinates into coordinates belonging to a operative reference frame. Calibration system has been designed and developed.

- Server side has been designed and implemented from scratch using a simple three-tier architecture running in a set of containerized microservices under a common virtual network. Security has been addressed, implementing a first token-based access control management, as well as three levels of security based on access control, authorisations and devices taxonomy.

- Integration between Server and HoloLens2 has been the most challenging part of the entire project. A protocol for collaborative mapping and data fusion (the D.U. algorithm) has been designed, implemented and tested using both API tests and automated tests. The system is able to merge data from different sessions in efficient way, as well as to merge data from different opened sessions, making possible a collaborative mapping and data exchange regarding the spatial informations.

- The D.U. algorithm has been integrated then in the client application, from the networking low level, to the integration of each API call, to end with a time-based orchestration for upload and download.

- Closed the implementation of the application, a subsequent analysis of the critical parts of the application allowed to find a set of Key Performance Indicators able to represent the performances of the application; a number of reports (e.g. CSV data extractions on the client side) have been implemented. Two report in particular: *Chasing Speed Metric Report*, based on assessing the performances of the localisation/mapping system; and *Client Stats Report*, able to capture the data exchange volumes between client and server. Also benchmarks and testing procedures have been created, in order to test the application.

- The work ended with a set of tests (mostly simulated) with the two objectives to estimate a good combination of basic parameters and to assess the overall performances of the system.

The github repository containing the final project and the results can be found at the following link:

- *github.com/programmatoroSeduto/SaR4HoloLens2*

The main contribution of this project has been the development of a more extended system able to centralize the information and to distribute it in real-time among different devices. HoloLens2 has been set inside a more complex organisation, starting from Search and Rescue practise.
Great importance have been given to the problem of mapping with simple means, with the idea to make a first simplified representation of the environment to add then the details. For instance, there are many affinities with the previous application and the data model proposed in this report: meshes produced by scene understanding could be "anchored" to the waypoints of the accessibility graph.

6.2 Device Limitations

Developing in HoloLens2 has been particularly challenging: sometimes it is not trivial to understand how to do something in the correct way, and a lot of tests are required to overcome lacks in the documentation. Here are the main limitations of the device and its development toolkit (MRTK2 with Unity)

- A developer needs to juggle in four different APIs to implement something: the Unity API (the base API for Unity), the MRTK2 API (graphical interface and subsystem abstractions, partial), C# (especially .NET) and Universal Windows Platforms (UWP, a extension of C# allowing the cross-platform development in one language). There are also a bunch of situation in which all the APIs contain methods to implement a feature, but only one of them is supported, so all the alternative have to be tested in order to understand the situation.

- Simulated environment and device environment are completely different in terms of supported APIs. The implementation has to take into account the double compatibility: in particular, Unity does not allow to execute UWP code, requiring either test projects in UWP applications (not trivial to build up) or direct tests on the device. Specific code has to be written two times: one time for Unity, and one another for the device, hence having two different pieces of code to debug. Moreover, basic resources are different: for instance, the version of C# in Unity is different with respect to the one provided by the device, leading sometimes to unexplainable incompatibilities of code which is perfectly working in theory. Debugging is very complex in this situation.

- Universal Windows Platforms enforces a twisted way of managing access to resources (e.g. disk storage): a code which is simple in common .NET requires a lot of extra statements in UWP.

- Usage of Inheritance is limited by Unity. Classes handled in GUI must inherit MonoBehaviour, avoiding to use general interfaces unless in the code. There are many other incompatibilities like this.

There are also some limitations related to the hardware of the device:

- Microphones have not a good quality: voice commands are often not perceived, and when they are heard, the processing spends almost 2 seconds to issue the command, which is a large response time, unacceptable in practice.

- Tracking losses arise, for instance, when the user passes through a door, or in narrow and chaotic spaces: in these situations, the device loses the odometry coherence, moving the local frame somewhere else and making immediately wrong all the subsequent measurements. Moving in large spaces instead does not produce tracking-loss issues; this strange behaviour has practically assessed during device testing.

- HoloLens2 does not support GPS geolocation different from IP location, which is highly not reliable. Unity API for IP location is not supported by HoloLens2, so UWP has to be used. In order to have reliable geolocation, it is required the integration with other devices with this capability.

In the end, HoloLens2 could do great things in terms of user experience, but developing on it is very challenging, and requires a lot of patience. Here a architecture running on HoloLens2 only is proposed, but I think the device could be exploited better distribuiting the workload between HoloLens2 (as tool for visualisation only) and another device such as a smartphone (integrated with a app developed in Android Studio, or also in Unity) or a PC (in this case, it could be evaluated a integration with Robotic Operating System).

6.3 Future Developments

This project is just the beginning of the journey. Many ways still can be explored starting from this base. Here is part of a long list of open points and future developments which could be interesting to put in practice:

- A final application rework could be a good idea before iterating again: there are some defects which can be corrected; and there are some pieces of code not employed, left in the project from previous experiments.

- More can be done in terms of visual features and assets. There are several proposals in the project not included in the final version for a simple matter of time: for instance, Unity has a navigation system which can be configured to work within a set of landmarks, so it can be created another visual, similar to the ones created in this version of the project, showing only the path to follow to reach a checkpoint.
  In conclusion, visuals can be improved a lot. Objective of the current work was just to povide development patterns to implement visuals. Finally, the lack of the possibility to set the app inside the app is currently a huge limitation.

- Adaptability of user interfaces could be highly improved with a better coding of the SAR Exploration Control Unit. It has not developed so much for a matter of time.

- The component *MinimapStructure* currently used for tracking the instanced virtual objects, also supports the possibility to hide a object instead of destroying it, allowing to save computational resources. Currently, virtual objects are destroyed, which is not the best approach. It should be modified the Drawer class in order to support a better resources management through hide/show instead of simple instanciate/destroy.

- A future development could consider a different architecture for the application at hardware level. Some operations are not good to perform in HoloLens2 due to a number of technical limitations in the APIs used for developing in such a device. Better performances could be obtained integrating the device with a smartphone, and moving part of the processing to the smartphone, using HoloLens2 only for visualisation. A local hotspot, starting from the smartphone, can be used to create a socket connection between the devices. It could be another promising development.

- At localisation/mapping level, a proposal based on tuning is discussed, which is not the best thing: in general, we would desire a system able to tune itself.

- At server level, it has been thought a separation between a staging area (a level of the HoloLens2 integration containing raw data mixed among sessions) and a quality area, supplied by a separated data processor able to create a clean data structure from the stating data. This part is entirely to implement.

- A better connection management is required by the hololens2 application. There are some problems when the connection is down: this should be a simple defect to correct. In general, it could be found a better way to close the application and then to close connections. In the final app it has been provided a button "close connection" since HoloLens2 does not call the unity callback `OnDestroy()` neither the class destructors, so currently any connection have to be explicitly closed.

- There are in the project automated tests to try out the implementation of collaboration, but it has not been tested in practical with two devices. Practical tests could be organized with two or more devices in different areas, to see how the applications works also in this case. Concurrency should be reviewed, since the current version of the server application has some weak points about this aspect.

- SceneUnderstanding is a important feature which could be re-introduced in future iterations of the project, giving to it he right place. It could also be used actively by the user, for instance associated with the RayTracing to point towards accessible points. Other alternative applications are possible for this kind of feature.

- It has been partially implemented also the possibility to enable the user to correct a position wrongly detected by the system. In simple words, thinking for example on a system tracking the position of the user with a fixed time period of 5 seconds (for instance), the position recorded from the system could not be perfect with respect to the space where it is. Having the possibility to visualize and building a object the user can manipulate in Augmented Reality allows the user to make corrections on the positions which are found not precise or correct. Also cases in which paths have a too much irregular shape can be improved with such a system. This have been partially integrated in the HoloLens2 project: the waypoint can be

bounded to a GameObject representing that position, and reading the coordinates of the object in realtime instead of the very first measurement. Noteworthy that it produces some complications in managing the propagation of updates between devices, hence such a feature have been discarded in this project: this has to be re-designed, trying to make a better draw of the desired user experience.

- Part of the first proposal was the integration of data from wearable sensors and HoloLens2 (as discussed in chapter 1; Semiconductor [2023]), which has been partially structured in this project but not provided in the final project. Having a operative frames system, it is possible to integrate any operative sensor, and to share informations with HoloLens2 through the server.

- Data model could be improved deleting all the fields and the tables not really used in this version of the project; starting then from a clean data mode, it is easier to detect how to work later on.

- A graphical interface (e.g. a web app) is completely missing right now for performing changes of settings and permissions in the database. currently, only direct SQL can be used for doing that. It could be provided a little webapp in order to be able to remotely modify the environment, or other solutions. This is a necessity for a real organisation, especially for coordinating the operations by the control center: for instance, let's think on the situation in which a new reference frame is agreed on the field. It has to be provided a method to define it quickly. Another case requiring a user interface is access control.

- Security Access controls could be managed better than in this implementation. They should be rewritten and documented, in order to have completely clear all the types of checks. Currently there are mainly three transactions able to perform security checks and operations. The system currently can detect some cases of potential security issue looking at logon operations and resources acquisitions, but a better work can be done performing a deepened security analysis. In future, the system could use this information to take actions against the suspect user, for instance blocking the account after a number of unsecure attempts.

- Current implementation of the Upload process makes a slightly wrong management of the waypoints staging table. The main problem lies in the request ID used to manage separated steps of the upload transaction; this does not allow to use rollbacks, which is one of the most useful feature in HoloLens2. This choice could result in a problematic behaviour in managing multiple processes as time, i.e. concurreny is limited. Transactions should be rewritten in order to not to place such a column in the staging table, writing results in the table only at the end of the transaction, and not before as intermediate step. More information can be found in the second Appendix (8).

- Referring to paths implementation, there are some significant drawbacks related to the use of primary keys to refer to the waypoints instead of local IDs: code

quickly grows in complexity in dealing with paths, unfortunately. The current implementation needs to be reviewed in managing this problem, which can generate redundancy and complications in the end.

- About the client side, the problem of areas separation has been partially addressed. To have a idea, let's consider the following practical scenario: the user, after some exploration of the environment, disables in some way the positions database, walks far from the previous positions, and then re-enables the positions database. In the current implementation, this sitation produces a long connection from the nearest found point, that could be very far from the current user's position.
  A first evaluated approach is based on dividing mapped waypoints in areas evaluating the distance from the nearest point: when a distance higher than a threshold is identified, the system produces a new area, identified by a integer index. Areas remain separated until the user doesn't pass from one area to another one (represented by a well-known frontier point): in this case, areas are joined, since a connection between the two area have been found.
  This is the general idea, implemented and tested in simulated environment and on client-side only; positions database still supports the identification of two types of insertions, in order to identify area. This approach have to be exploited and implemented better in future iterations of the project, also providing server-side support. For a matter of time, this feature have been discarded in this version of the project. It could be reintroduced, or also changed with a better one if available.

Chapter 7

Appendix 1 - Final Application Overview

Summary

Two are the projects implemented under the scope of this work: the client application, running on HoloLens2, and the server application, running on a remote machine accessible through static IP address (more informations in chapter 4 and in appendix 2, 8). A variety of tools and languages have been empoyed in order to build up such a architecture. Both the parts of the projects have been developed from scratch, from the architectural design and guidelines making, towards the coding part, ending with the testing (mostly in simulation) introduced in the next section of this chapter.
Let's start talking about the client side.

7.1 The Client Side

Here's a small introduction about all the tools the developer needs to explore in order to be able to build something in HoloLens2; the following bulleted list reports the most relevant ones:

- Unity, and the Unity API/Framework, is the main game engine for this project
- C# is the main language employed by Unity, and Visual Studio is the main development IDE for the code
- Universal Windows Platform (UWP) is a API in C# required sometimes to talk directly to the device; in fact, there are some functionalities not wrapped in the Unity API, and for these we're forced to re-implement them using this kind of API. The action of reading/writing files, of primary importance for logging for instance, requires the usage of UWP. UWP can be seen as the lowest-level part of the C# development process.

- Mixed Reality ToolKit v2 (MRTK 2) is the main official framework by Microsoft, written in C# for Unity (and not only) which allows to make easier the development process with HoloLens2, at least for what concerns the management of features such as inputs (i.e. voice commands, gestures), direct diagnostics, scene observer, and many other settings. It provides a large set of packages specifically made for HoloLens2 aiming at simplifying the overall management of the application. It is particularly useful to develop experiences on HoloLens2.

HoloLens2 supports three development modes: the way using Unity (our way), the one using Unreal Engine 4, and the last one which is the native development using OpenXR and C++ directly against the DLLs of the device. We did choose Unity to keep a continuity with the previous project proposed by Testa [2022].

A unity project is essentially made of GameObjects (i.e. instanced graphical assets), Assets, Components (the code) and Scenes. In particular GameObjects and components are the most important parts of the project:

- A *Component* (AKA *scripts*) is a functional entity; it can be intended as a node which starts running at a certain moment, and continues updating frame by frame. Many components can run "in parallel", combined to create a network of components and, in the end, the logic of the application. It is implemented as a C# class endowed with a `Init` method, called when the component is spawned for the first time, and a `Update` method, called at each frame only if the object is active.

- A *Game Object* is a item, either visible or not, having a transform (position, rotation, scale) and relative connection with other game objects; Unity organizes game objects in a tree-like structure. Each game object can contain a number of components. A component, to be able to work, has to be assigned to a particular game object and instanced. In Unity, a object can be applied to a Game Object only if it is a MonoBehaviour, or inherited from that class; common classes cannot be attached to game objects.

All the game objects and components are contained under a *Scene*, i.e. the virtual space where everything happens. Unity offers the possibility to run at the same time different scenes, merging the content to create a third scene from the combination of those two.

A scene follows the rules of the classes as well: a scene is instanced. Every object and component in Unity follows the OOP standard.

7.1.1 HoloLens2 App Structure and Guidelines

At development level, one of the limitations of Testa [2022] were to not to have a clear project organisation and clear guidelines to support *reusability of the code* and *segregation of the functional parts* of the application. Hence, a set of guidelines have been introduced; many of them are related to the operative code management.

Project scripts are mainly divided in two big sets: package scripts, and custom script. *Package scripts* are well-defined functional units, which are re-usable in other project simply moving files. Installing them could be very simple, given some possible dependencies with other packages.

Custom scripts are project-specific and can put together different packages in order to create more complex behaviours related with the aim of the project. Package scripts are more general than custom scripts.

Each functional unit may have at least three types of scripts inside: components (the script that can be instanced as MonoBehaviour), utils (script not attachable to Game Objects, static objects, or data structures) and module testing (raw scripts made for testing the main functional units of the packages). At the beginning of the project, also the 'type' has been considered, but, due to some significant restrictions in Unity regarding the usage of interfaces, this kind of script has been discarded in subsequent implementations.

Dependencies are generally allowed among packages only. But, *a package script cannot depend on a custom script*: this is a fundamental guideline. A package script can only depend on another package script.

There's a third fundamental type of scripts, called *Project scripts*, providing project-specific implementations but also general services which can be used by the package scripts. This is in part a violation of that main guideline: since a project script is a sort of custom script, it should be not allowed to have a package script using a project script, but due to the generality of the functionalities offered by the project scripts, a violation of that rule is allowed in this case.

In the end, there are the custom scripts, provided at scene level. To speak more precisely, since also a scene is a form of custom package, scripts are organized in this way: starting from the root of the project (Assets folder),

- Package scripts are contained under Assets/Script/_Packages/<*name of the package*>/<*type of the script*>
- Project scripts are contained under Assets/Script/<*type of the script*>
- Custom scripts are related to the scenes, so contained always under a folder like this: Assets/Scene/<*scene name*>/Scripts/<*type of the script*>

To make clear the dependencies, script are enforced into namespaces following the below convention:

- Package script are located under namespaces labeled Packages.<*name of the package*>.<*type of the script*> where the type is one of the types discussed above: Components, Utils, ModuleTestings.
- Project scripts are located under Project.Scripts.<*type of the script*>; notice that the namespace is intentionally generic, in order to allow packages to use the components inside without referring to specific names if not necessary.

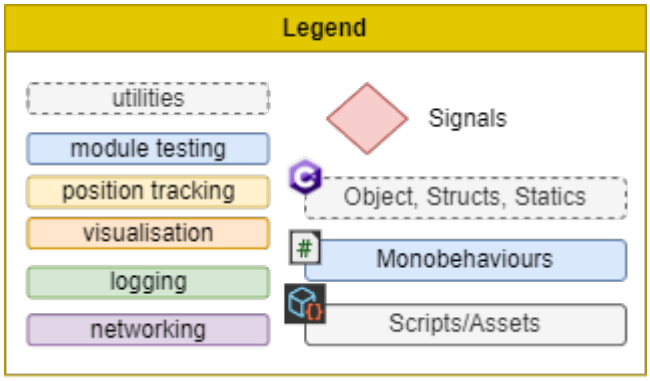
- Custom scripts are collected along with the scenes: Project.Scenes.<*name of the scene*>.Scripts.<*type of the script*>. A custom script can refer to everything inside the proejct, even other scenes if needed.

Graphical assets are generally located in the *Assets/Resources* folder, which is a special folder for Unity: for instance, to be able to spawn a complex item from a model (prefab), Unity requires that the prefab is located under that folder. Aven in this folder, reusability of the assets has to be the very first aim when organizing the material. In particular, the current Resources folder is made up of paths like this: Assets/Resources/<*Specific Name of the graphical Asset*>/<*type of asset*>. Prefabs are located directly under the root of the specific name, whereas other items, such as materials, are located in slef-describing folders under the specific name.

7.1.2 Available Main Packages

Let's talk more specifically about the services implemented inside the project, starting from the packages. Each package provides a set of services for a particular aspect of the application, from networking to position tracking to end with visualisation, assets management and storage services. For more coherence with the way the system is designed in a Unity-based environment, 7.1 shows the meaning of the symbols used later. Three packages are discussed here: the network integration, the positions database, and the visual controller. Other packages are discussed in the next sections.

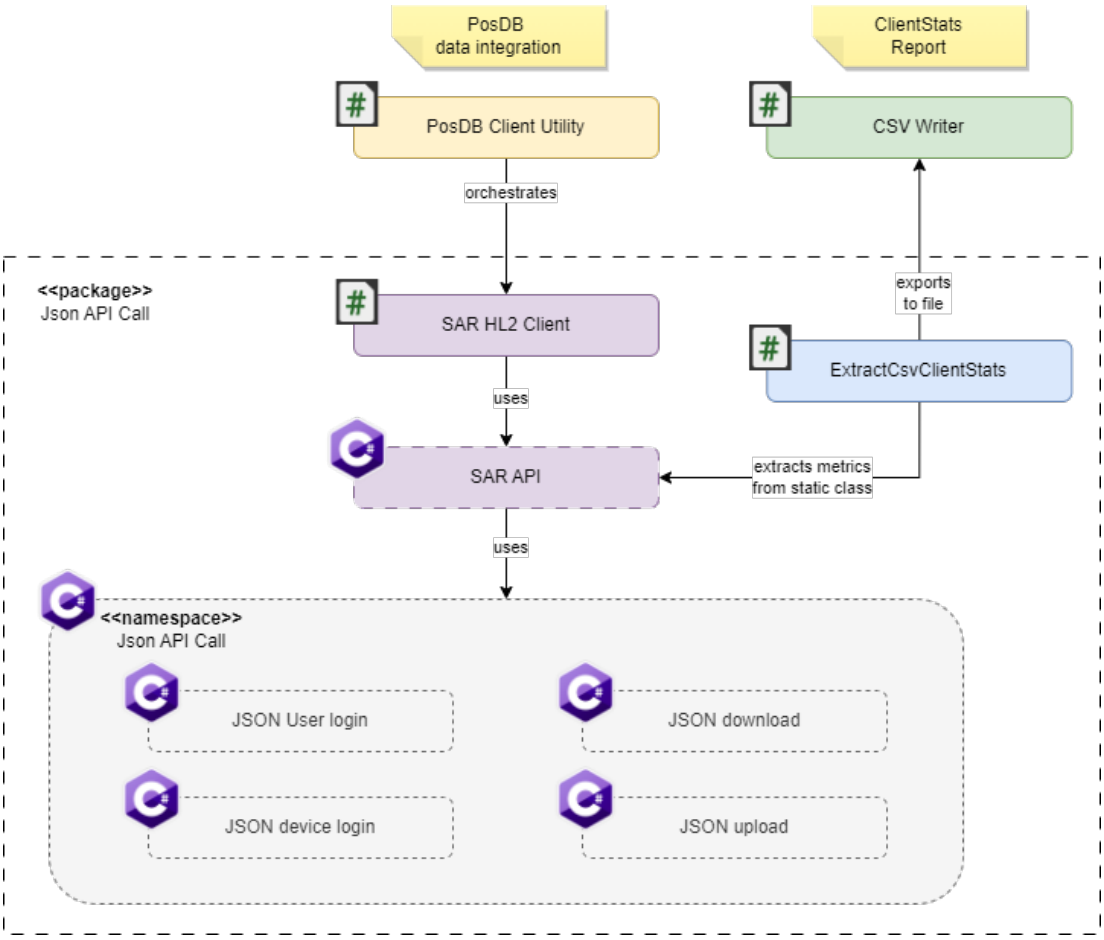
Figure 7.1: Meaning of the symbols in the component diagrams in this chapter



SAR4HL2NetworkingServices is the package containing the services used for communicating with the server. So far, it is a package specifically implemented for the integration of the current API running on the server. It is structured as a stack, where each block encloses the functionalities belonging to one particular level of abstraction, from the direct API call management to the orchestration. Figure 7.2 represents the internal structure of this package.

The networking package is mainly a passive set of services, implementing coroutines able to handle the server at various level: the *Sar HL2 Client* offers the main processes

Figure 7.2: HL2 Package for Networking



at medium level, whereas xthe static class *Sar API* provides low level implementation of methods, one for each API endopoint. *Sar HL2 Client* combines the endpoints and performs marshalling in order to simplify as much as possible the operations of the *PosDB Client Utility* which is part of the Positions Database Package.

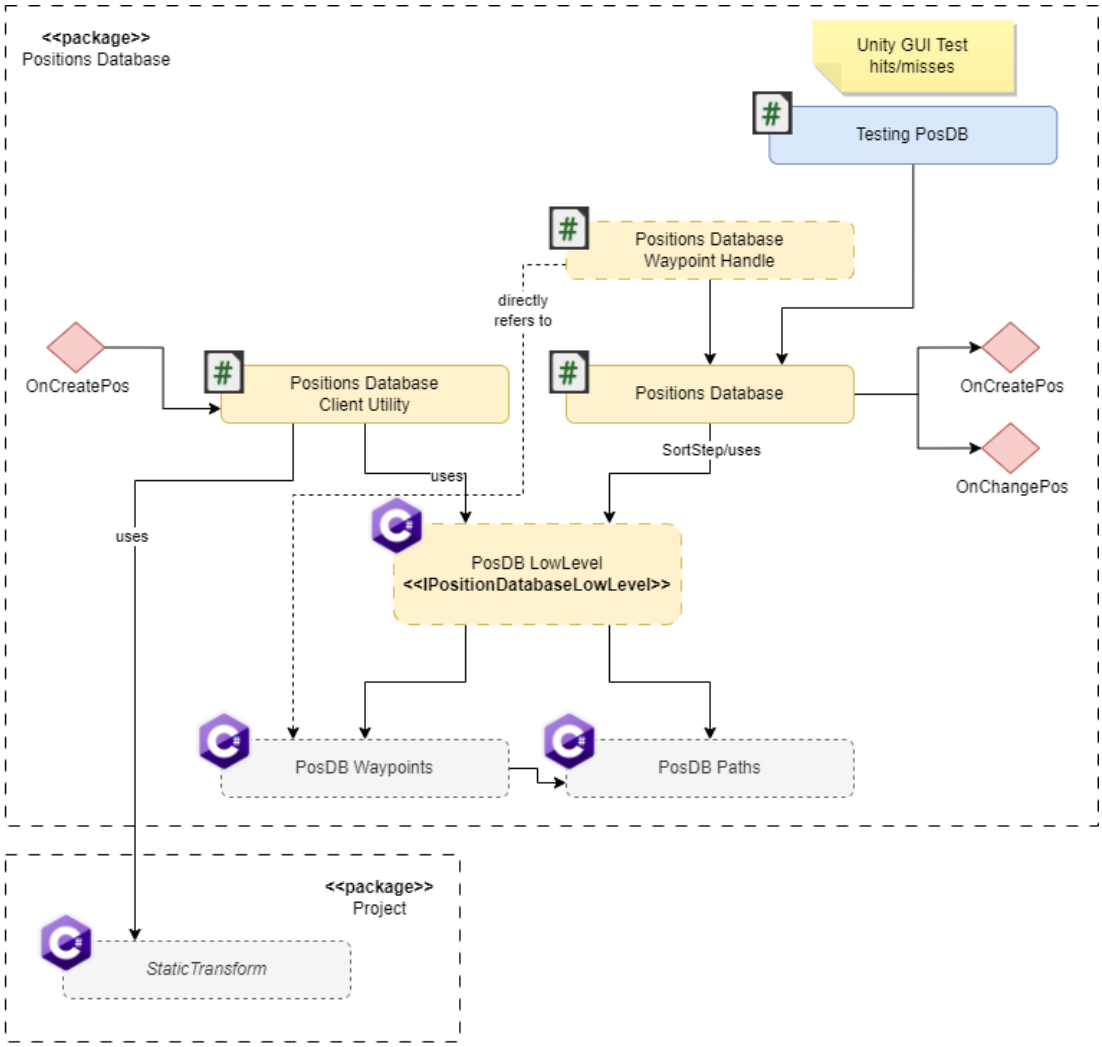
Important to say that *Sar API* manages also the credentials as well: the static class is able to keep track of the access tokens used for communicating with the server. Also service statistics are collected by this component, encoded in a particular data structure, ready for any class asking for service metrics.

The last two elements are *ExtractCsvClientStats* and *JSON API Call*. The *ExtractCsvClientStats* is the component generating the Client Stats report, and the namespace *JSON API Call* contains json-serializable classes representing the formats of the requests and responses for communicating with the server.

PositionsDatabase is the package providing all the main functionalities for inter-

nal position tracking and mapping. It has a complex internal organisation, as shown in figure 7.3. Most of the elements inside have been explained in detail in chapter 3. The classes waypoints and paths represent the basic building blocks of the mapping data structure, shared across all the modules inside the package and beyond (to keep the representation simple to understand, only the most relevant links are shown).

Figure 7.3: HL2 Positions Database Package

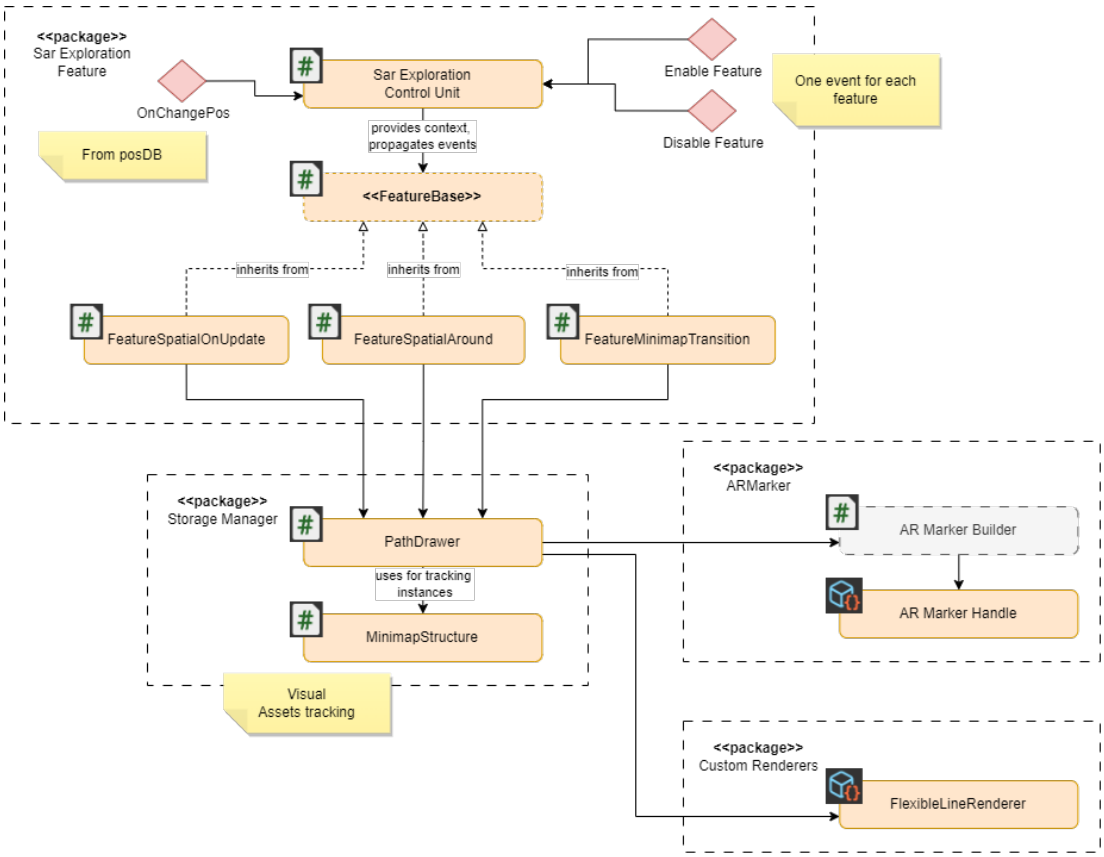


PosDB Low Level, implemented satisfyying the C# interface *IPositionDatabaseLowLevel*, collects both the tracking procedure (based on dynamic sort, as explained in chapter 3) and the storage methods needed to keep the data structure in memory at runtime. It is meant to be shared, in order to improve efficiency, and i nfact this implementation shared the object with both *Positions Database* and *Positions Database Client Utility* that orchestrates the coordination with the server

side, as explained before. *Positions Database* has many important functions: first of all, as a Unity Monobehaviour endowed with Update cycle and instantiatable, it keeps ordered the low level calling the `SortStep()` at each frame after having obtained the user's position; it implements the creation of new positions depending on the user's position, enabling the system to map new points; and in the end, it implements a set of signals. Two in particular: the `OnCreate`, issued when a new position is discovered, and `OnUpdate` issued when the current position changes. Notice that, since the `OnCreate` also changes the current position, the `OnUpdate` is issued along with the previous signal.

SaRExplorationFeatures is the most important part of the visuals, to be considered in conjunction with the package **StorageManager** providing components to manage the storage of the visual assets in the virtual space. Ad represented in figure 7.4, the current visuals implementation is a set of 4 different packages; this structure has been created with the future perspective of creating a more rich set of assets with a common, standard structure.

Figure 7.4: HL2 Visual Packages



The *Sar exploration Control Unit* is the main component collecting all the endpoints to pilot the visuals. Currently each feature has their proper set of standard methods for

enabling, switching or disabling visuals from simple events and with parameters which can be set from GUI. The component sets the environment based on the visual. It listens also for the signal `OnChangePos()`, coordinating the update of the visual based on this instead of performing a simple coordination based on `Update()` Unity callback: it allows to obtain a lighter processing. Notice that graphical processing can be heavy, so it should be avoided to update every graphical interface frame by frame.

The component interacts with features through a "interface" called *FeatureBase*: properly speaking, it is not a true interface, but a MonoBehaviour never instanced and made to be inherited by other components able to perform the job in some way. This class simply contains the common traits, required by the controller to act a general control process, in particular regarding the feature switch. Internally, right now the controller has a very simplified structure, but in future interactions it could support a more advanced way of managing resources according to the external inputs.

As said in the chapter 2, three visuals have been developed here:

- the *Breadcrumbs visual* is implemented by the `FeatureSpatialOnUpdate` component
- the *Surrounding visual* is implemented by the `FeatureSpatialAround` component; visual depth is provided by the main controller as part of the execution context
- the *Minimap visual* is a bit more complex: the transition between full-size and minimap is performed by the `FeatureMinimapTransition`, but the internal update of the data into the visual is performed "re-applying" the `FeatureSpatialAround` component

More complex visual features can be implemented using this way of reasoning: for instance, to implement a static version of the minimap, showing a static big box with the paths inside, it is sufficient for instance to create another feature inheriting `FeatureBase` and to modify the controller, making it able to handle the new type of feature. Also SceneUnderstanding and Unity-based navigation can be integrated in this way, as well as more challenging combinations of tools.

The controller also manages the instances of the features, making all the complexities totally invisible to the end user and to the other components using the controller. It passes a execution context to all the FeatureBase-type components, which includes a direct reference to the positions database, and another handle which is a direct reference to a `PathDrawer` component, from another package. It hides the complexity of spawning, hiding and destroying virtual objects in the virutal space, simplifying a lot the implementation of the Features components. The `PathDrawer` is a tool: it allows the user to "draw a point somewhere" or to "draw a line from somewhere to somewhere else" for instance; many other operations can be implemented with a simple modification of the component. For example, currently the component supports two main assets: the so-called "AR Marker" (representation of a waypoint) and a "flexible line

renderer" (called in this way since the basic line renderer in Unity does not move as the extrema move; as shown in the graph, it is a script, but related to visualisation more than to the functional part of the system). Let's consider the situation in which we're extending the set of available assets, for instance introducing another type of marker; this can be added simply creating a class representing it, and to handle it modifying the `PathDrawer` in a way such that it supports the usage of two different markers; a idea could be to provide a method for changing "point drawer" padding from one type of marker to another one, and re-using the previously defined interface to draw with the other marker. And other developments like this can be put in action using such a approach.

Notice that the marker asset is not built directly, since it has a complex structure: building it directly requires a unsustainable amout of code to the PathDrawer. To avoid repetitions and confusion in coding, the asset can be built using a service class called `ARMarkerBuilder`. There could be also the necessity to distribute the build process on many frames, due sometimes to some limitations by Unity in calling the `Init` for the object, generating strange errors, difficult to solve; all these details can be hid using such a service class type. Currently, the Line Renderer can be created with no problems, but in future also that asset could have a builder.

Another important point is that the PathDrawer has the need to manage efficient management of resources. Each asset is associated to a code (currently based on the stable key) in a lookup table (i.e. a dictionary). To make more efficient such type of operations, a so-called class `MinimapStructure` has been implemented, able to track instanced game object, and to "confund" the operation of destroying a object and to hide a object in efficient way. It is essentially a dictionary, but storing object in a way such that some particular visuals are supported.

For instance, one of the features implemented in this component is the possibility of storing objects ordered by one numerical dimension. One of the first visuals proposed in this project but not included in this version of the project has been to implement a static minimap, representing a set of objects distributed in the space (for instance, the content extracted from SceneUnderstanding) along the Z axis, and inclusing a big "moving plane" along the Z axis, able to hide everything outside the space of the plane, so a tool for making 2D projections of the 3D map. This is exactly the work for which the MinimapStructure has been created times ago: it not only allows to keep all the game objects ordered (using a ordered insertion based on the dimensional value) and to track which are visualized and which are not, but also to select in a very efficient way the index where to find the item with a method like this: assuming that the distribution of the elements is almost uniform, and keeping the minimum value of the criterion as well as its maximum value, the index of the item located around a certain Z value is the one which index is approximately around the value $\frac{z-min}{max-min} \cdot len(array)$, allowing to take, most of the times, the correct index, with almost no search; it could be found also a way to "balance" the probability distribution if the data are not uniformly distributed between the minimum and the maximum. There's still the method implemented in the class, able to perform such a operation upon a range of values. This minimap structure

can be combined with other instances of this class in order to create more complex filters, for instance the possibility to filter both Z and X in the map. Figures 7.5 and 7.6 give a idea of what has been done in this case.

7.1.3 Signals and Events

Due to a important limitation of Unity Events which does not allow to change the parameters of the event instance, signals are generic empty calls. `UnityEvent` contains the reference of the object and the function to call, but it cannot carry data since they are not changeable in the caller, so the called object has to keep also a reference to the object which is able to perform the call. 7.7 represents the situation. Parameters of the call can be changed only from GUI: this is the big deal.

A types-based approach have been evaluated, trying to avoid usage of UnityEvents with the aim of simplifying the event generation to the simple function call: it can be done using interfaces. But the solution have not been considered satisfying for two reasons: the code becomes rapidly more complex, and Unity does not support interfaces used in GUI. Common interfaces are hidden drom the GUI: only classes inherited from MonoBehaviour are shown, which is a big practical limitation. So, the solution found uses the UnityEvents only to issue the signal, leaving to the called class the task to call some getter or public method or field to retrieve the informations related to the event. Calls are synchronous, hence, from the caller point of view, it is enough to prepare the information to pass to the called class immediately before the `Invoke()`: all the classes will be called under the same frame. This has the drawback to create a delay in the `Update()` phase, especially when the objects to invoke is high and each function requires time to execute. A workaround consists in keeping as brief as possible the called method, eventually using coroutines to distribute the processing time in the other frames in parallel to the caller runtime.

Despite this, components are created and destroyed in the current implementation, addressing the problem of fine memory management in other iterations of the project.

7.1.4 Project-specific Scripts

Almsot all the scripts in the project, at least the "modern" ones, refer to other classes contained not in the Packages zone, but among the Project scripts. Aim of the proejct scripts is to provide more flexibility on the basic functionalities of the components, maybe already present in Unity but not managed as we'd prefer. For instance, the case of the message prints from the application: sometimes we want to have a better way to select what to print, feature that Unity does not allow by default.

In other cases, functionalities can be implemented as project script in order to guarantee a shared access endpoint for that feature: for instance, the calibration can be handled in this way. The main idea behind the project scripts is this: if we were thinking of creating a standard solution template as basis for other projects, project scripts would have to be part of that solution template.

Figure 7.5: Sliding tool before the activation. In the image, the SceneUnderstanding map is "simulated" by a set of small random 3D shapes in the space of the minimap, i.e. under the same root.

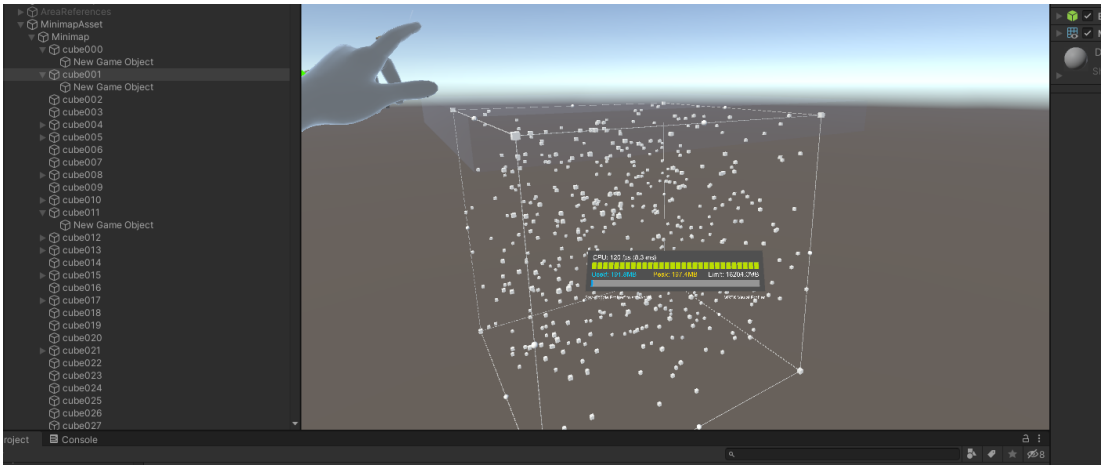


Figure 7.6: Sliding tool after the activation. The sliding tool was a quasi-plane cube: the user can move it along the choosen dimension, in this case the Z axis. When enabled, all the other shapes are hid, except for the ones falling into the range of the tool.

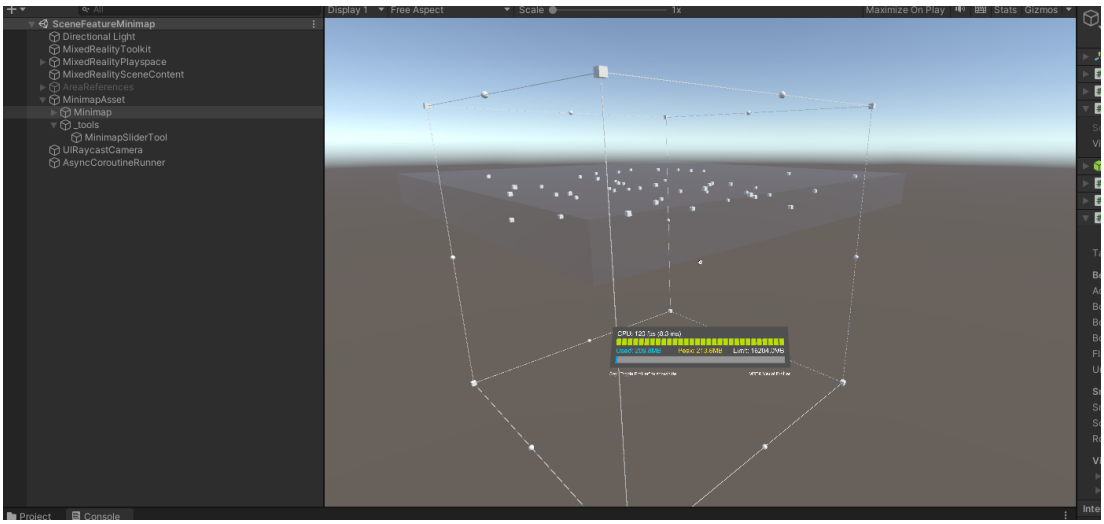
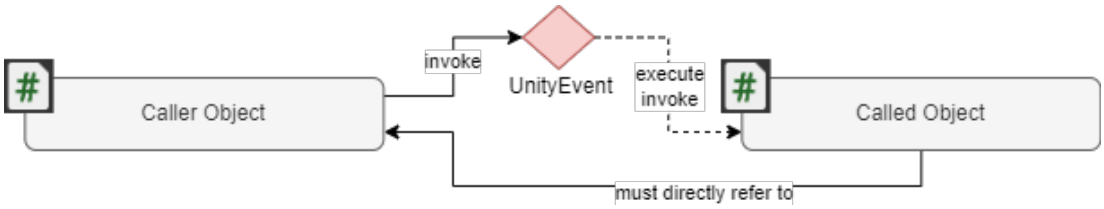


Figure 7.7: Signals Pattern represented. The UnityEvent is a object containing all the references to the called object. Notice that the entire process happens in the same frame for each called component.



Static Classes are perfect for sharing functionalities across the system to whatever other component. The only limitation is that the static classes cannot access the frame-specific informations, leading to some design complications to obtain those data. For instance, user's position can be accessed only by a component: a static object does not have the context to access it, even if maybe the call is accepted by Visual Studio.

The project contains mainly three static objects:

- the **StaticLogger** contains a enhanced interface to interact with the Unity output capabilities from code. Please refer to the logging section for more details about the logging system.
- the **StaticAppSettings**, along with the project component **AppSettings**, offers a functionality similar to the ROS *parameter server*, allowing to exchange informations between the components as global parameter. Unity does not offer such a functionality.
- the **StaticTransform** offers a shared way for accessing calibration informations, and in particular to transform vectors between "absolute" frame and local frame. The Positions Database client utility, for example, uses this feature immediately before the export phase, or the import phase

The project exposes also other components, used for a general management of the application runtime and startup, listed below:

- **ProjectMonoBehaviour** is a class inheriting directly from MonoBehaviour, adding custom functionalities to each component in order to support a project-specific way of handle instanciation of components. One of the problems solved by this kind of component is, for istance, the need for understand if a component is ready to work; currently Unity does not support a suitable flag able to notify when the component is ready, so it has been implemented in the project behaviour, also including a function used for notifying when the component is ready.
- **ProjectAppSettings** is mostly a project-specific script, allowing to make a complete configuration of the project in just one place: it is a control panel to be used

via Unity GUI for quickly configuring all the components of the project. It should be present in every project following the standard hereby presented

- **EntryPoint**, currently not used, provides the possibility to change scenes and to load and compose them, very useful to build a user interaction. Not used here since the user aspect have not been developed much here.
- **CalibrationUtility** is the base component of the calibration implementation, providing all the frame-based data to the static object (it is a sort of backend for the static object)
- the last one is the **StartupScript** which leverages the **ProjectMonoBehaviour** to start scripts in a precise order. It is very useful in particular to overcome a limitation in Unity about the custom execution order of the script; to keep more predictable the system startup,

7.1.5 Logging and Data Collectors

We already discussed the static object *StaticLogger*, providing useful features for filtering log lines and for enabling or disabling log directly from inside the code. This function also allows to control the behaviour of the simulation, eventually pausing it when something happens, which can be used for instance to implement breakpoints in very simple terms.

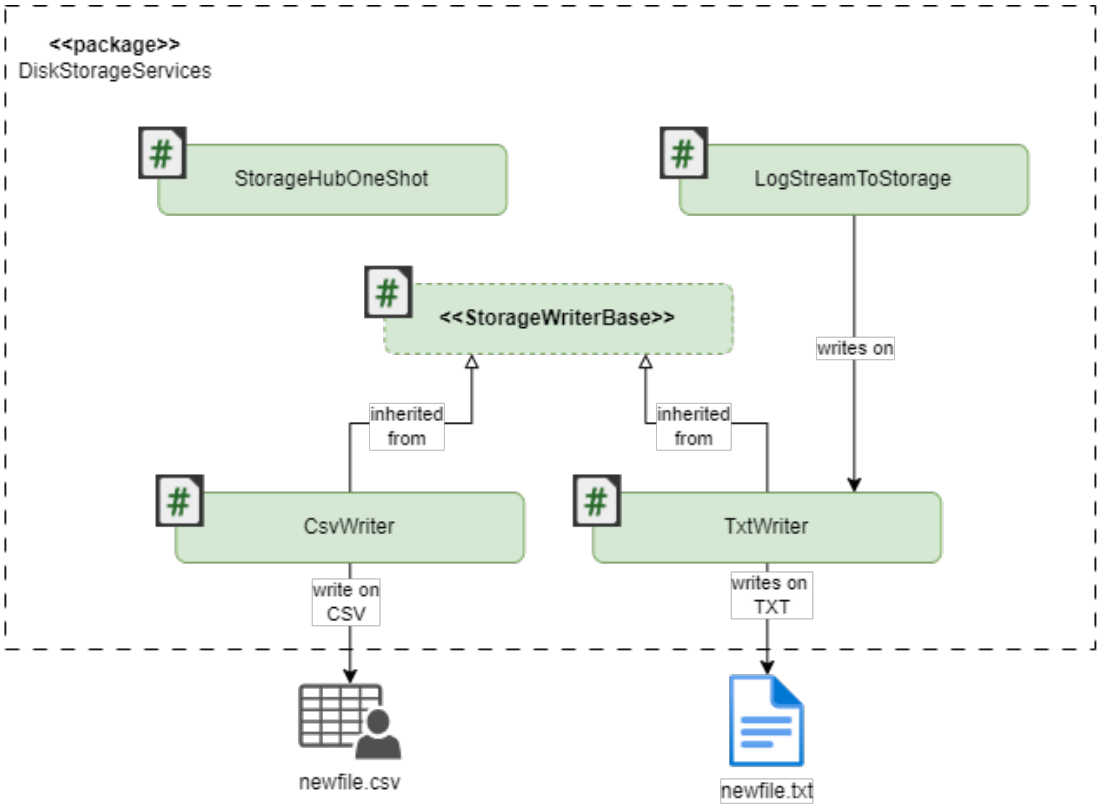
Logging has, strategically speaking, primary importance, since beyond the simulation it is really difficult to understand what happens under the hood without any feedback from the application. And even more important, it is almost impossible to understand a crash from the device itself: HoloLens2 does not provide a system log detailed enough for understanding precisely the problem when a problem occurs. So, this kind of feature is completely in the hands of the developer.

Another need is to extract data and metrics from the application. For instance, this project makes wide use of CSV log files to obtain performances evaluations and other data. All these things have to be implemented along with the project.

There's a package called *DiskStorageServices*, containing components to implement streams and other operations on the disk. 7.8 gives a sintetical representation of the package structure. It provides at least two ways to transfer data between the program and the storage: in streaming (`CsvWriter`, `TxtWriter`, only write supported), and in "one-shot" (`StorageHubOneShot`, supporting both reading and writing).

Talking about streaming channels, one component represents exactly one channel opened on a file. `StorageWriterBase` is the main interface implementing the core functionality. based on UWP API since .NET is not allowed by the device, and there's no way for writing files directly from Unity API. As development approach, since the disk writings in streaming can easily saturate the device, a double process implementation is provided: the first one collects messages in a queue; the second one takes messages from the queue and write them distributing the task on many frames. This is the core functionality hidden in the `StorageWriterBase` class. At high level, the class provides

Figure 7.8: Package Schema: DiskStorageServices



a simple method `Write()` writing whatever string on file without worrying about string formatting.

The problem of the precise file formatting is moved in components inheriting the Base one. Currently the project supports two file formats: `TxtWriter`, which is a trivial inherited class from the base one; and `CsvWriter` which has special methods to handle the CSV format, providing a reasonably simple set of parameters at Unity GUI.

`StorageHubOneShot` provides functionalities both to read and to write a simple text file: also in this implementation, just strings are handled, leaving the format check to a higher class. IThe structure is almost the same of the StorageWriterBase, except for the fact that the file is closed immediately after the action: so, the component is reusable at runtime.

In the end, `LogStreamToStorage` collects all the messages from the Unity log and transfers them as stream towards a txt file. It is extremely useful to know what's going on inside the application, as well as a analysis tool for detecting bugs and unexpected situations.

7.1.6 App Complete Arch and Startup

Figure 7.9 represents the most complete set of blocks created so far. Some blocks can be removed before the deployment on the device: for instance, some report can be disabled in order to save resources. Only functional blocks are represented, hiding the services such as most of the static classes and the blocks managing the boot of the application.

To set on such a structure, the procedure to follow is roughly resumed below (mainly divided into two steps):

1. first of all, TXT file is created
2. then the LOG stream to storage, in order to record the log from all the other components
3. other file streams are opened in this phase (the green ones) and file readings if needed
4. SarClient is called, ready to create the connection
5. PosDB Client Utility is calleed; the first step immediately after the Init is the connection to the server through SarClient; the component waits for calibration, since otherwise the data from teh first download cannot be transformed in local coordinates
6. exploration control unit is called; it creates one Path Drawer and one Minimap Structure, but no features yet

The application is now waiting the calibration. The user calls it through the voice command, follows the procedure, and calibration data are collected. The startup procedure continues:

1. the calibration utility passes data to the Static Transform, making it available to all the following transformations
2. then, it enables the positions database
3. the positions database starts running and collecting informations from the user's position
4. in the meanwhile, the client utility issues the API command *download* and downloads the first data from the server; downloaded data are integrated directly at low level by the client utility

In test environments, data extractions for the reports are enabled immediately after the first download. The system is fully enabled after this phase, and visuals are usable, since the system is updated by events from the positions database.

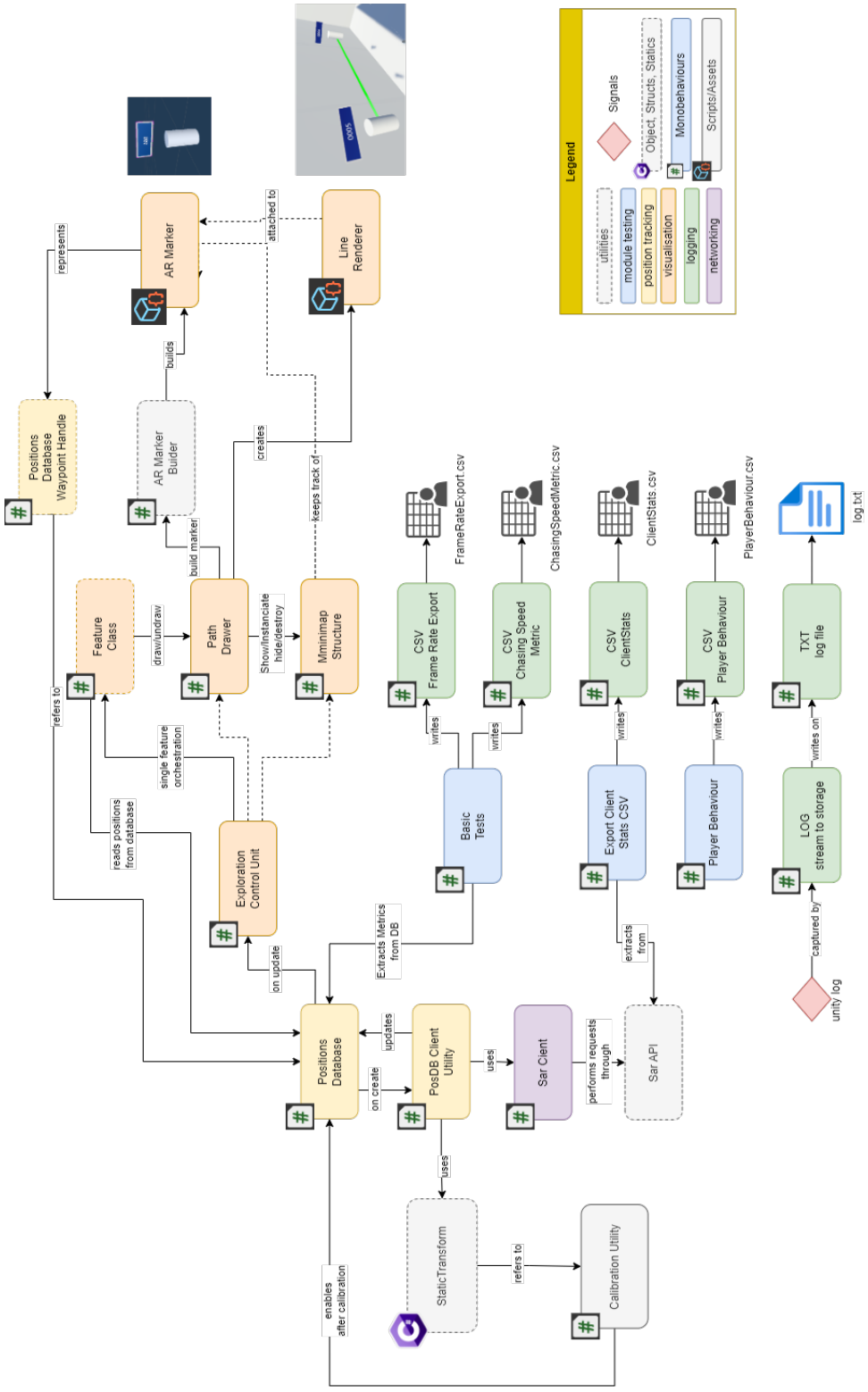


Figure 7.9: Overall HoloLens2 System Architecture

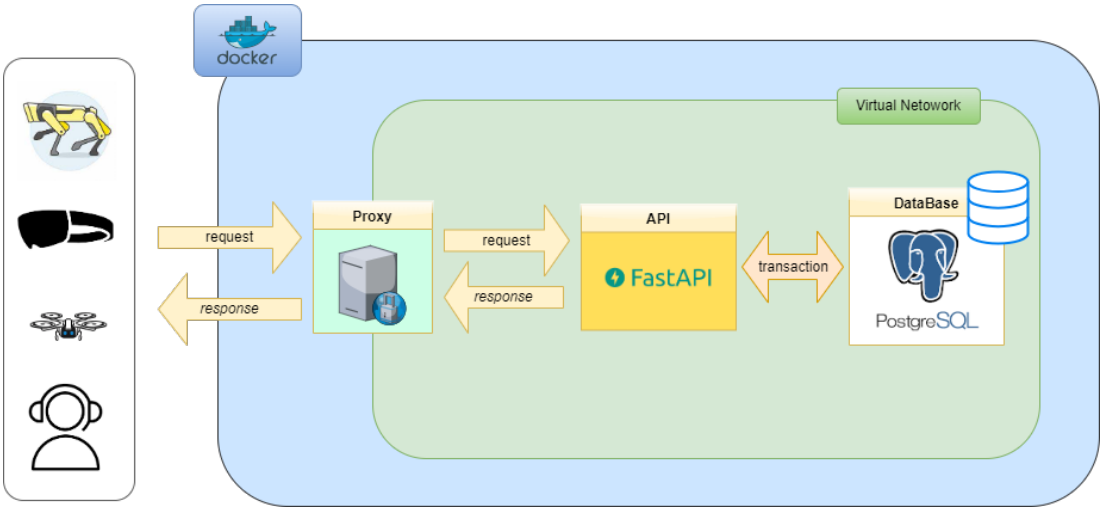
7.2 The Server Side

After discussed the structure of HoloLens2, let's start talking about the implementation of the server. Chapter 4 describes a bit the implementation of the server without going into details too much; here a more technical point of view is provided on the implementation.

7.2.1 Development Tools and Architecture

The figure 7.10 shows a overall picture of the internal architecture of the server. The entire architecture runs as a set of independent microservices; all the structure is based on Docker, an open platform for developing, shipping, and running applications Docker [2023] supporting containerization and able to provide a framework for bringing together a set of microservices, as done in this case.

Figure 7.10: Overall Server Architecture



The core of the server is enclosed inside a virtual network, isolating the database from external accesses not passing through the proxy server, implemented as a container; it runs a NGINX traffic manager Nginx [2023] able to redirect requests to the API, implemented as another container based on FastAPI, a well-known, versatile API frameowrk FastAPI [2023], a lightweid framework to implement HTTP-based APIs, easy to learn and to use. The main part of the server is the database, implemented using PostgreSQL PostgreSQL [2023], another well-known Relational DataBase Management System with a cozy SQL syntax, which can extended in order to support queries involving vectors (geoqueries). This implementation uses a version of PostgreSQL extended with *PgVector* PgVector [2023].

7.2.2 Database Implementation Structure

About PostgreSQL, there are literaly thousand of ready-to-use images online containing a installation of PostgreSQL. I did choose here a version of the container which alreasy integrates the package `pgvector`, a package allowing to perform geoqueries, i.e. SQL queries based on distances, in a efficient way. It adds new operators specific for the computation of distances.

The container, when it is started for the first time, has a script which is its entry point, in charge of performing the first configuration. This entry point is able to execute custom scripts as well, when the microservice starts for the first time.

Three scripts have been created for the first setup in the scope of this project:

- **setup_datamodel.sql** contains the definition of the data model of the database
- **setup_libraries.sql** contains custom functions to support database transactions (see the API in the next section)
- **setup_experimental.sql** can be useful for testing new features: it contains custom functions and data definitions in testing phase

Moreover, the current implementation can load a bunch of samples in the database, as initial set of informations when the server starts up for the first time.

7.2.3 API Container Structure

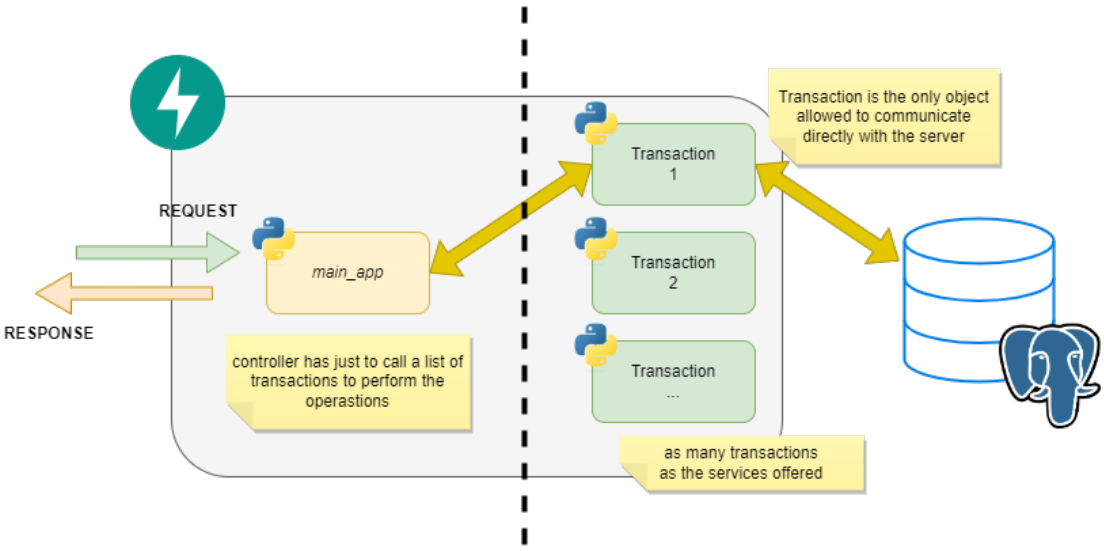
The API is entirely contained in one microservice (container) called 'main_app' implementing all the logics and the endpoints, making a bridge between the clients and the data. It has been implemented using the FastAPI framework (API endpoints modeled as Pydantic models), running on a Uvicorn web server, and using Psycopg2, a simple connector from the API to the database running on another container. It is a very classical architecture for these types of systems.

In figure 7.11 there's a brief schema summarizing the structure of the implementation hereby proposed.

The idea is simple. The main app is just a interface, mapping the request to the transaction able to satisfy the request, but it never operates directly against the database. Transactions, which are simple classes built using a common structure, implement the set of SQL queries and logics to really perform the requests. All the system is build using the principle of modularity:

- each API endpoint is written always with the same structure with a standard, to simplify the implementation of new possible endpoints
- each API endpoint uses a number of transactions, which are simple classes, hence a developer can implement a new transaction simply implementing a new class with a particular structure, and finally to integrate it with some other API endpoint

Figure 7.11: API Transactions Architecture



Important to say that the main_app script starts up the container as well, loading configuration settings by the environment; the project simplifies also the management of new parameters with a set of auxiliary object a developer can use to build a complete environment starting by the Dockerfile used for building the container.

Each transaction, as much as possible, follows a simple working schema like the following one:

1. CHECK Phase : the class checks if there are the conditions to perform that operation
2. EXECUTION phase : the class performs operations and logging, according to the results obtained in the CHECK phase. In case the CHECK failed, the EXECUTION phase will write the error on the system log instead of performing the execution of the transaction against the database.

Since a transaction is just a class, building the response given a request, it can be easily handled by a entry point without particularly complex logics, as much as possible and depending on the situation. Notice that the only phase performing updated into the database is the EXECUTION phase: the CHECK one never updates the database.

Here's a list of currently available endpoints:

- `(GET) /api/` : this endpoint is used just to check either the service is available or not
- `(POST) /api/user/login` : a single transaction for user login; access key and approver user are required; session token is provided in case of successful call

- `(POST) /api/user/logout` : a single transaction for user logout; session token must be provided in order to logout
- `(POST) /api/device/login` : the user tries to acquire a resource, i.e. a device; session token must be provided, as well as the reference to the device the user wants to acquire; simple confirm is sent back
- `(POST) /api/device/logout` : the user tries to release a resource; session token must be provided
- `(POST) /api/hl2/download`, HoloLens2 specific: the device is trying to download positions around from the server; two tokens are required (session token and 'based_on' token).
- `(POST) /api/hl2/upload`, HoloLens2 specific: the device tries to load positions into the server; two tokens are required to perform the operation; keys negotiation is sent back

For more details, please give a look at the life cycle of a user/device, and at the D.U. algorithm, all explained in chapter 4.

7.2.4 System Startup

The project provides two main "entry points" to start the project:

- **Simulated start** to be used for checking che correct behaviour of the system as well as to perform development; proxy server is not run, and the database is accessible using a simple connection from any database tool given access credentials; database is not persistent, since there's no volume mounted for storing data
- **"Production" start** from the main DockerCompose file; it also includes the proxy server, and the database is not accessible; database storage is persistent thanks to a mounted volume

It is possible to try out the project and to develop by using the first type of start, provided by the `main_app/compose.yml` file. Further information can be found in the technical documentation attached along with the project, in form of a tutorial: these documents explain how to correctly start the project, as well as the settings, and finally how to configure/install/uninstall the project on a remote server machine.

Chapter 8

Appendix 2 - Server Side Technical Details

Summary

In this additional chapter, the reader can find more technical details about database implementation guidelines, implementation of the Download Upload procedure, and implementation of the alignment algorithm.
Also limitations of the currently implemented approach will be discussed here, to be addressed in future iterations of the project.

8.1 Naming Conventions

Names into the database are formulated in a way such that, from a first look, it is clear at least which is the type of a table, or which is the general type of a column. As general rule, all the SQL identifiers are *uppercase*, *alphanumeric*, *separated by underscore*.

About the names of the tables, the name is always built with this rule: a letter identifying the type of the table (fact/dimension/lookup), followed by one underscore, ending with the identifier of the table. In particular,

- `F_` indicates a *fact table* which (in simple terms) collects a set of measures with timestamp. Log tables, waypoints tables, are all examples of fact tables
- `D_` indicates a *dimendion table* which collects attributes and characterizations for entities involved in the organization; a dimensional table describes the entity, and a fact table describes what the entity does
- `L_` indicates a *lookup table* which is a simple association between two set of informations. It is typically a lightweid table, similar to a dimensional one as function

About the name of the columns, the rule is the following: the identifier of the variable, followed by one underscore, ending with two letters which represent the type of the variable.

- _FL indicates a *BOOLEAN value*, a column with binary behaviour (notice that NULL is not excluded a-priori)
- _VL indicates a *NUMERIC value*; in simple words, a temperature, a component of a vector, numbers
- _TS indicates a *DATETIME value*, a complete timestamp with date and clock time
- _DT indicates a *DATE value*, a simple calendar date with year month and day
- _HR indicates a *CLOCK TIME value*, which is clock time without date
- _DS indicates a *STRING value*, a textual value or a set of data encoded in bytes
- _ID indicates a *IDENTIFIER value* which can be either a string o a hash code; ID indicates a code
- _PK indicates a *PRIMARY KEY value*, the primary key of the table
- _FK indicates a *FOREIGN KEY value*, a link to another table

When possible, the model have been kept intentionally simple, trying to use just the most basic types to avoid "strange behaviours" and errors due to datatype mismatch.

In the scope of this project, it has been preferred to keep a format also for the codes used by the organization. Here are the formats:

- **user ID** : 'SAR_ID' + *10-numbers numbers sequence* + '_USER'
- **device ID** : 'SAR_ID' + *10-numbers numbers sequence* + '_DEVC'
- **reference frame ID** : 'SAR_ID' + *10-numbers numbers sequence* + '_REFP'

The database supports some metadata, but right now the topis has not been exploited too much. In general, some tables tabe into account at least the time when a record is created, as well as if it is still valid. But, currently, there's no process yet enabling someone to create or delete a user for instance, for the sake of simplicity.

8.2 Fake Token Protocol

Download transaction is the most important in this implementation of the protocol because it not only allows to receive informations from other devices, but also because the download transaction checks-in the client in the staging table.
Origin of the reference frame is a special point: if the client is authorized to publish data into the staging tables, it must exist a record referred to the origin of its operative reference frame, containing the association between the current session ID and the inherited session ID. A client which is not registered in this way, cannot exchange data

at all with the system.

Referring to the high level way of work of the download transaction, the client has to provide two tokens to exchange informations: its current session token ID (due to the successfully done login) and the inherited token (received after the first download). A attacker interested in altering data of the staging table has to "steal" two codes.

But, it exists one case where only one token is required, i.e. when the current session ID is promoted as first token of a dataset. It could lead to security risk since it is sufficient to send a packet with one token and to try to access the server with a brute force attack. To enforce the usage of two token by all the devices, and to support other possible situations like this, the fake token protocol has been implemented, which consists on a simple mapping from a real token (that could be empty) to a fake one. Even a empty token could have a fake one.

All the fake tokens are stored into a table available in this implementation with the name `sar.F_SESSION_ALIAS`. Here are the base informations required to check which client owns the token:

- `DEVICE_ID` : the device owning the fake token
- `USER_ID` : the user owning the fake token
- `OWNER_SESSION_TOKEN_ID` : (*not null*) a fake token is generated under the scope of a currently opened login
- `USER_SESSION_TOKEN_ID` : the token to hide with a fake token

It is important to make as much specific as possible the client owning the token, to make more detailed checks. The principle is that if the attacker steals the fake token, it needs a lot of informations to access the system, and all these informations must be correct; this makes more difficult to understand which are the correct credentials.

The token to hide can be NULL as well, meaning that it is required just one token to have a credential to ask to the client: information can be not required by the data model, but it should be required anyway by the system during the security checks before any operation.

When the user logs in to the staging table, the system creates a fake session token in this way:

1. A `SALT_ID` is created, which is a random HASH code made to add entropy upon the final fake token
2. `OWNER_SESSION_TOKEN_ID` and `SALT_ID` are combined and hashed again to create the final fake token

The query for generating the code is similar to the following one (using prepared statement):

```
SELECT
    %(user_id)s AS USER_ID ,
    %(device_id)s AS DEVICE_ID ,
    %(owner_token)s AS OWNER_SESSION_TOKEN_ID ,
    data_tab.session_to_hide AS USER_SESSION_TOKEN_ID ,
    data_tab.salt_code AS SALT_ID ,
    MD5(
        CONCAT(
            data_tab.salt_code,
            COALESCE(data_tab.session_to_hide, 'NOTINHERITEDSESSION'),
            data_tab.salt_code
        )
    ) AS FAKE_SESSION_TOKEN_ID
FROM (
    SELECT
        MD5(FLOOR(RANDOM()*100000000)::TEXT) AS salt_code,
        %(user_token)s AS session_to_hide
) AS data_tab
```

This is another credential, in terms of security of the API, and it must be provided by the client each time a certain operation is required. The server could require different fake tokens, one for each resource the client wants to control. Finally, fake token is provided by the server only once.

To check if a client has access, after generated the fake toke, the system goes immmediately to the table of fake tokens trying to "map back" the fake token to the real one, needed for accessing the resource table. In this case, the row is unique only if device, user, owner token and fake token are correct; otherwise, a query SELECTint these fields will produce a empty result, meaning that either the credentials are wrong or the user is not registered at all; in both the cases, the access is denied to the resource. Here's a query implementing this check (prepared statement):

```
SELECT
    USER_SESSION_TOKEN_ID, FAKE_SESSION_TOKEN_ID, SALT_ID
FROM sar.F_SESSION_ALIAS
WHERE 1=1
AND USER_ID   = %(user_id)s
AND DEVICE_ID   = %(device_id)s
AND OWNER_SESSION_TOKEN_ID = %(owner_token)s
LIMIT 1;
```

Noteworthy that the staging table of waypoints uses only real tokens, and the client always must provide its real session token. The system could query the staging table to check if the user is already registered for publishing results into that table. If the user is present in that table, it is logged in, and the provided fake token is correct, only in this case the client is authorized to publish new data.

There are several ways to use the datamodel of the project to provide a more detailed error checking. A basic authentication method is provided in this work using fake token protocol, to be improved in future iterations. Currently the protocol has been designed to support only one resource acquired, which is not realistic: the data model here has to be updated to support also a identifier of the resource, allowing the protocol to manage one token per resource required. Currently there's no way to manage such a case.

8.3 D.U. Protocol - Download Transaction

The download endpoint returns new data to the devices, extracting them from the database and selecting them in a way that no redundant communication is performed; it is responsible also to mix informations and to spread them among the devices.

Download transaction is structured in a first step, finding near accessible unknown waypoints from the dataset, and a second step for paths extraction on the basis of the data from the first step; in the end, a last step allows to obtain data in the final form, even tracking which data have been made available to the client.

Important to say that the Download transaction is the first that the client has to call when it starts working: this is the reason of the name of the protocol. Download also perform session check-in.

8.3.1 High Level Interface

The download entry point is `/api/hl2/download`, which expects a JSON package similar to the one showed below. Among the informations provided by the client, *center* is the current position of the device, and *radius* is the radius of the request. In the example below, the client is located at the origin of its frame, and asks for waypoints and paths within 10 meters from its position.

```
{
    "user_id": "SARHL2_ID8849249249_USER",
    "device_id": "SARHL2_ID8651165355_DEVC",
    "session_token": "c8f85700b6f29a7a6841842349340724",
    "based_on": "",
    "ref_id": "SARHL2_ID1234567890_REFP",
    "center": [
        0.0,
        0.0,
        0.0
    ],
    "radius": 10.0
}
```

Important to notice that the field `based_on` is set `NULL` in this example, meaning not that the user session defines a new dataset, but that the client is asking to be registered inside the staging table: it is a check-in request.

The server will search a minimal and coherent set of points and paths: in other words, either a subgraph or a complement to attach to the frontier to a well-known and shared graph between client and server. Sesponse will have this simple format:

```
{
    "timestamp_received": "2023-10-22T15:22:09.311227",
    "timestamp_sent": "2023-10-22T15:25:44.900443",
    "status": 202,
    "status_detail": "success",
    "based_on": "393c5561dbdef2f990ccc015b7f8d0af",
    "ref_id": "SARHL2_ID1234567890_REFP",
    "max_idx": 8,
    "waypoints": [
        {
            "pos_id": 4, "area_id": 0,
            "v": [3.69, 0.0, 3.69],
            "wp_timestamp": "2023-10-22T17:24:51"
        },
        {
            "pos_id": 6, "area_id": 0,
            "v": [5.54, 0.0, 5.54],
            "wp_timestamp": "2023-10-22T17:24:54"
        },
```

```
        ...
    ],
    "paths": [
        {
            "wp1": 4, "wp2": 3, "dist": 1.30
            "pt_timestamp": "2023-10-22T17:24:51"
        },
        {
            "wp1": 3, "wp2": 2, "dist": 1.30
            "pt_timestamp": "2023-10-22T17:24:49"
        },
        ...
    ]
}
```

Max Id is the number of points inside the dataset at the time of the request. It will be used by HoloLens2 to generate IDs which are more likely to be accepted diretly during nogotiation in the UPLOAD requests.

8.3.2 Waypoints extraction

Let's proceed step by step. As said, the first step is the extraction of the waypoints from the database given the distance. It only extracts the points which are inside the distance and unknown by the client. Here the main points of the processing.

First of all, it is possible to obtain the points that the client knows by simply SELECTing the session the client has already opened. Each session is identified by a single session token ID. The query is like this:

```
SELECT DISTINCT
    F_HL2_QUALITY_WAYPOINTS_PK,
    LOCAL_POSITION_ID
FROM sar.F_HL2_STAGING_WAYPOINTS
WHERE SESSION_TOKEN_ID = '8418cc...'
```

Notice that these are also the set of waypoints previously known by the device.
It is also possible to obtain all the points inside the same dataset: they are all the ones belonging either to the "based_on" session or belonging diretly to that code. The result will contain points in part known and in part unknown by the device.

```
SELECT DISTINCT
    F_HL2_QUALITY_WAYPOINTS_PK,
    LOCAL_POSITION_ID,
    SESSION_TOKEN_ID,
    SESSION_TOKEN_INHERITED_ID
FROM sar.F_HL2_STAGING_WAYPOINTS
WHERE NOT(wp.ALIGNMENT_TYPE_FL)
SESSION_TOKEN_ID = '8418cc...'
AND (
    ( SESSION_TOKEN_INHERITED_ID IS NULL
    AND
    SESSION_TOKEN_ID = '-9237e7...'
    )
    OR
    SESSION_TOKEN_INHERITED_ID = '-9237e7...'
    )
```

with a operation of exclusion of one set from another one, it is possible to exclude all the points from this result which are known, obtaining the set of unknown points. The final step is to choose, among the unknown points, the ones near enough to the client. This will be the final result, for what concerns the first processing of waypoints. All these steps are enclosed in PostgreSQL functions returning subqueries, and combinded together. This way of reasoning spreads informations from the other device to the one is requiring new data if any, creating a form of collaboration in mapping.

Noticeable that this set of points is for sure *overflowing*: it contains unaccessible points due to the fact that the client, with its set of known points, simply could not have discovered all the paths linking them. This creates a anomalous situation, addressed later on with a bit more of analysis.

8.3.3 Paths extraction

As second step, the transaction performs a extraction of paths from the table, which are yet unknown by the client. It is performed in only one query:

1. The starting point is the table `sar.F_HL2_STAGING_PATHS`
2. From here, the query select only the paths within the radius around the current position which contain at least one unknown point (it is build upon the previous one)

The result will contain not only the paths from A to B, but also the paths from B to A, since the data model is a un-oriented graph: the final table will contain both the types of paths to make easier some other operations performed later.

8.3.4 Extraction filtering

The server should send only the strictly necessary points accessible by the device and near enough, and only if yet unknown. Links must exist at least upon the set of points already known by the client *plus* the ones the server is going to return.

It could happend sometimes that the path linking one position to another one is too long to not be entirely included in the extraction, leading to a unlinked set of informations. This created some troubles on the HoloLens2 application, so it has been implemented a mechanism to exclude not linkable waypoints.

The transaction currently extracts both the known and the unknown points in two separate data structures. Paths, known points and unknown points are used as input of an algorithm able to detect waypoints and paths to use. The principle is simple: starting from the current position of the client, and given the set of paths, a unreachable position cannot be reached using that set of path. A recursive algorithm has been implemented to perform such a analysis.

First step is data preparation. Here, all the waypoints are collected from the current session as a set of local IDs. The purpose is to link every possible point from the known ones to the unknown ones, and to exlude at the end of the process all the points not linkable to the known ones through the paths found inside the distance. Procedure 14 shows how this operation is performed.

To use this algorithm, the transaction extracts all the waypoints from the database known by the client, and applies this algorithm to all the points: considering just the corrent position of the client is not enough, since there could be a path which is near to the client and accessible through some other point not accessible with the paths inside

Algorithm 14 Path Exclusion Procedure

```
function PathAnalysis(current_wp, wp_set, pt_set)          ▷ three sets of IDs
    found_wp_set ← empty set
    for all wp in wp_set do
        tup ← (current_wp, wp)                              ▷ both local IDs
        if tup in wp_set then
            found_wp_set.add(wp)                  ▷ Found a reachable Waypoint
            pt_set.remove(tup)                              ▷ not to be excluded
        end if
    end for
    if len(found_wp_set) == 0 then
        return wp_set, pt_set                               ▷ Nothing reachable
    end if
    for all wp in found_wp_set do
        wp_set.remove(wp)                          ▷ reachable, try to remove
        wp_set, pt_set ← PathAnalysis(wp, wp_set, pt_set)
    end for
    return wp_set, pt_set
end function
```

the initial set extracted at the previous step (notice that only the known paths are extracted).

8.3.5 Data collection and response

The final step of the download transactions accomplishes different purposes:

- The points and the paths the server is going to return will become known by the client, hence information have to be tracked inside the session data
- Data preparation of the response (trivial)
- Exclusion of waypoints and paths not accessible

PostgreSQL allows to handle JSON in queries in a simple way: this has been employed for easily integrating exclusion list in that query. Another interesting feature is to combine INSERT statements with SELECT statements, which allowed to update two tables in the same query.

Here's the structure of this final step, simplified:

```
WITH
exclusion_list_wp AS (
SELECT DISTINCT wp
FROM JSON_POPULATE_RECORDSET(null::json_schema_wp, {...excluded wps...})
)
, exclusion_list_pt AS (
SELECT DISTINCT wp1, wp2
```

```
FROM JSON_POPULATE_RECORDSET(null::json_schema_pt, {...excluded pts...})
)
, pre_insert_wp AS (
SELECT
    ...
    wp.LOCAL_POSITION_ID AS LOCAL_POSITION_ID,
    ...
    wp.UX_VL,
    wp.UY_VL,
    wp.UZ_VL,
    ...
FROM get_unknown_waypoints_in_radius(...) AS wp_base
LEFT JOIN sar.F_HL2_STAGING_WAYPOINTS AS wp
ON (wp_base.F_HL2_QUALITY_WAYPOINTS_PK = wp.F_HL2_QUALITY_WAYPOINTS_PK)
WHERE wp.LOCAL_POSITION_ID NOT IN (SELECT wp FROM exclusion_list_wp)
)
, insert_wp AS (
INSERT INTO sar.F_HL2_STAGING_WAYPOINTS (...)
SELECT * FROM pre_insert_wp
)
, pre_insert_pt AS (
SELECT
    ...
    WAYPOINT_1_STAGING_FK,
    WAYPOINT_2_STAGING_FK
FROM get_unknown_paths_in_radius(...)
WHERE (
    WP1_LOCAL_POS_ID, WP2_LOCAL_POS_ID)
    NOT IN (SELECT wp1, wp2 FROM exclusion_list_pt)
)
INSERT INTO sar.F_HL2_STAGING_PATHS (...)
SELECT * FROM pre_insert_pt
RETURNING TRUE
```

8.4 D.U. Protocol - Upload Transaction

Main purpose of the upload transaction is to load new data from the device into the staging tables and make them available to the other devices. The main part of this complex object is the implementation of the *alignment algorithm* as well as the *IDs negotiation*. Paths and waypoints are integrated in two different steps.

8.4.1 High Level Interface

The API provides the entry point `/api/hl2/upload` which expects a request like the following one, just to provide a little example. Notice that the user and the device have been autenticated, and a `based_on` token have been already provided by the first download.

The foolowing JSON snippet represents a typical upload request, complete of waypoints and paths in a suitable format:

```
{
"user_id": "SARHL2_ID8849249249_USER",
"device_id": "SARHL2_ID8651165355_DEVC",
"session_token": "8418ccf470288a81bc0f02437eb7a40e",
"based_on": "450d5df1bfe07b239dfb933920147e24",
"ref_id": "SARHL2_ID1234567890_REFP",
"waypoints": [
    {
        "pos_id": 4,
        "v": [
            3.6970508098602297,
            0.0,
            3.6970508098602297
        ],
        "wp_timestamp": "2023/10/22 17:24:51"
```

```
    },
    {
        "pos_id": 5,
        "v": [
            4.621275901794434,
            0.0,
            4.621275901794434
        ],
        "wp_timestamp": "2023/10/22 17:24:52"
    },
    ...
],
"paths": [
    {
        "wp1": 5,
        "wp2": 4,
        "pt_timestamp": "2023/10/22 17:24:52"
    },
    ...
]
}
```

The response for such a request, when it succeeds, it is code `200` with a package returned like the following one:

```
{
    "timestamp_received": "2023-10-22T15:22:09.311227",
    "timestamp_sent": "2023-10-22T15:22:09.980186",
    "status": 200,
    "status_detail": "success",
    "max_id": 93,
    "wp_alignment": []
}
```

There are two important fields in this response: `max_id` which is a integer, and `wp_alignment` which is a list of "dictionaries".

`max_id` notifies HoloLens2 that the maximum ID of the local IDs for that dataset is that value. This means that HoloLens2 must propose new indices which should be *greater than* that value, since every value less than `max_id` is already taken by another position. This makes more likely that a request ID becomens a local ID without renaming.

It can be obtained from the database, waypoints table, using a query like this:

```
SELECT
MAX(LOCAL_POSITION_ID) AS MAX_LOCAL_POSITION_ID
FROM sar.F_HL2_STAGING_WAYPOINTS
WHERE SESSION_TOKEN_INHERITED_ID = '<based-on-token>'
```

In particular, the value is computed *after the new points are inserted if any new point have been found.*

Let's say there's the need for rename. `wp_alignment` will contain something like this:

```
{
    "timestamp_received": "2023-10-22T15:22:09.311227",
    "timestamp_sent": "2023-10-22T15:22:09.980186",
    "status": 200,
    "status_detail": "success",
    "max_id": 51,
    "wp_alignment": [
        {
            "request_position_id": 4,
            "aligned_position_id": 51
        },
        {
            "request_position_id": 5,
            "aligned_position_id": 11
        }
    ]
}
```

With a message like this, the server is telling HoloLens2 to change labels to these points with the ones in the package, to keep aligned the two data sources.

8.4.2 SQL Waypoints Processing

The first items to be processed when the transaction execution starts are the waypoints. In particular, the processing is implemented in only one SQL query, taking request data in JSON format, and performing table inserts. The query returns a alignment table at the end of its execution, possibly empty if there's no need for renamings. To process data, a sequence of common table expressions have been created.

PostgreSQL allows to convert JSON data into the content of a subquery using a sintax like this, combined with the possibility to use the cursor to perform a prepared statement:

```
    DROP TYPE IF EXISTS json_schema;
    CREATE TYPE json_schema AS (
         pos_id int,
         area_id int,
         v vector(3),
         wp_timestamp TIMESTAMP
    );

    WITH
    request_data AS (
         SELECT DISTINCT
              pos_id,
              area_id,
              v,
              wp_timestamp
         FROM JSON_POPULATE_RECORDSET(NULL::json_schema,
         '[
------------{"pos_id":1,"area_id":0,
------------"v":[0.0,0.0,1.305335521697998],"wp_timestamp":"2023/11/01 16:27:30"},
------------{"pos_id":2,"area_id":0,
------------"v":[0.0,0.0,2.611337184906006],"wp_timestamp":"2023/11/01 16:27:31"},
------------...
--------]')
    )

    -- SELECT * FROM request_data;
```

`json_schema` is a object used only for telling Postgres which is the format of the JSON package; it is dropped at the end of the processing.

Now that data are available for processing, it is needed to compare them with the data inside the dataset. We know the inherited session token, so we can select all the points to match with the ones in the request. The query will be like this:

```
    WITH
    -- request_data AS ( ... ) --> data from JSON request

    , session_data AS (
         SELECT
              F_HL2_QUALITY_WAYPOINTS_PK AS align_with_fk,
              LOCAL_POSITION_ID AS pos_id,
              LOCAL_AREA_INDEX_ID AS area_id,
              to_vector3( UX_VL, UY_VL, UZ_VL )::vector(3) AS v,
              COALESCE(WAYPOINT_CREATED_TS, CREATED_TS) AS wp_timestamp
         FROM sar.F_HL2_STAGING_WAYPOINTS
         WHERE 1=1
         AND NOT(ALIGNMENT_TYPE_FL)
         AND U_REFERENCE_POSITION_ID = 'SARHL2_ID1234567890_REFP'
         AND (
              ( SESSION_TOKEN_INHERITED_ID IS NULL AND SESSION_TOKEN_ID = '9237e74...' )
              OR
              SESSION_TOKEN_INHERITED_ID = '9237e74...'
         )
    )
```

```
-- SELECT * FROM session_data;
```

In the CTE above, taking into account that selection of the reference frame is useless in this implementation, the query makes a selection of data inside the dataset, which are not matched each other, hence the kernel of the dataset, made up of a set of measures from the very first session for the dataset (inherited session is NULL and session has the based_on token), and another parts of data from the other sessions inheriting from the base one. This is the data we have to use for matching.

The following comparison could have been implemented in code, but SQL is more efficient in doing this kind of operations:

```
WITH
-- request_data AS ( ... ) --> data from JSON request
-- , session_data AS ( ... ) --> kernel from database

, cross_data AS (
    SELECT
        request_data.pos_id AS req_pos_id,
        session_data.pos_id AS loc_pos_id,
        ( request_data.v <-> session_data.v ) AS dist,
        request_data.v AS req_v,
        session_data.v AS loc_v,
        session_data.align_with_fk AS align_with_fk,
        request_data.area_id AS req_area_id,
        session_data.area_id AS loc_area_id,
        request_data.wp_timestamp AS req_timestamp,
        session_data.wp_timestamp AS loc_timestamp
    FROM request_data
    LEFT JOIN session_data ON (1=1)
)

-- SELECT * FROM cross_data ORDER BY req_pos_id, dist, loc_pos_id;
```

This step is formely a cross join between the data from the request and the ones from the kernel available ultil the request. All the vectors from one side are matched with the other vectors from the other side, this is the most complex step in the query. Let's say the request contains 5 points, and the kernel contains 60 points: the query generates 150 rows, each of them with the distance from one of the points in the request and one from the kernel. *If the point matches with another one, the distance will be runder the threshold for at least one point.* But before detecting points, the query performs some other analysis over the data:

```
WITH
-- request_data AS ( ... ) --> data from JSON request
-- , session_data AS ( ... ) --> kernel from database
-- , cross_data AS ( ... ) --> compute distances

, analysis_data AS (
    SELECT
        req_pos_id,
        loc_pos_id,
        align_with_fk,
        req_area_id,
        loc_area_id,
        req_v,
        loc_v,
        cross_data.dist AS dist,
        CASE
            WHEN cross_data.dist IS NULL THEN FALSE
            WHEN cross_data.dist <= th.threshold_vl - th.toll THEN TRUE
            WHEN cross_data.dist > th.threshold_vl + th.toll THEN FALSE
        END AS WP_IS_REDUNDANT_FL,
        CASE
            WHEN cross_data.dist IS NULL THEN 100.0
            WHEN cross_data.dist <= th.threshold_vl - th.toll
                THEN ROUND(max_of(
                    ((th.threshold_vl - cross_data.dist) / th.threshold_vl)::NUMERIC,
                    0::NUMERIC ) * 100, 2)
```

```
            WHEN cross_data.dist >= th.threshold_vl + th.toll
                THEN ROUND(max_of(
                    (1 - exp( '4.75'*(th.threshold_vl - cross_data.dist) ))::NUMERIC,
                    0::NUMERIC ) * 100, 2)
        END AS QUALITY_VL,
        req_timestamp,
        loc_timestamp
    FROM cross_data
    LEFT JOIN ( SELECT
        '1.2'::FLOAT AS threshold_vl,
        '0.025'::FLOAT AS toll
    ) AS th ON (1=1)
)

-- SELECT * FROM analysis_data ORDER BY req_pos_id, dist, loc_pos_id;
```

First of all, each row previously generated is labeled with a flag, `WP_IS_REDUNDANT_FL` which states wethere the distance is below the threshold. Please notice that the "threshold" here is the sum of two parameters: the so called `threshold_vl` which is the radius around the point, and another parameter `toll` which is a tollerance around the critical value. The main idea is that, too much around the value of the radius, the situation is unclear; it could happen to a number of reasons.

The second conditional statement is the computation of the quality of the approximation. It evaluates the matched case with a linear quality formula, otherwise a exponential formula is employed.

The last step is just to extract results of the analysis: which points match, and which ones are really new with respect to the kernel at disposal. This is done by means of window functions. For each request position id (from the request), there are as many comparisons as the size of the initial kernel. Let's take one of them: there will be many rows, which we can set ordered by distance in ascending order. Doing so, we can understand either the point matches or not by just looking at the very first row after ordering: if it has a distance which is less than the threshold, it matches, otherwise it is a brand new point. The label is already computed, so the row is ready to take. The same operation is preformed for any request ID in the table at the same way: the result is the set of request points aligned and classified.

```
WITH
-- request_data AS ( ... ) --> data from JSON request
-- , session_data AS ( ... ) --> kernel from database
-- , cross_data AS ( ... ) --> compute distances
-- , analysis_data AS ( ... ) --> add match label and quality evaluation

, classification_data AS (
    SELECT DISTINCT
        req_pos_id,
        FIRST_VALUE(align_with_fk)
            OVER ( PARTITION BY req_pos_id ORDER BY dist ASC )
            AS align_with_fk,
        FIRST_VALUE(loc_pos_id)
            OVER ( PARTITION BY req_pos_id ORDER BY dist ASC )
            AS align_with_loc_pos_id,
        FIRST_VALUE(req_v)
            OVER ( PARTITION BY req_pos_id ORDER BY dist ASC )
            AS req_v,
        FIRST_VALUE(loc_v)
            OVER ( PARTITION BY req_pos_id ORDER BY dist ASC )
            AS loc_v,
        FIRST_VALUE(dist)
            OVER ( PARTITION BY req_pos_id ORDER BY dist ASC )
            AS dist,
        FIRST_VALUE(WP_IS_REDUNDANT_FL)
            OVER ( PARTITION BY req_pos_id ORDER BY dist ASC )
            AS WP_IS_REDUNDANT_FL,
        FIRST_VALUE(QUALITY_VL)
            OVER ( PARTITION BY req_pos_id ORDER BY dist ASC )
            AS QUALITY_VL,
```

```
            FIRST_VALUE(req_timestamp)
                OVER ( PARTITION BY req_pos_id ORDER BY dist ASC )
                AS req_timestamp,
            FIRST_VALUE(loc_timestamp)
                OVER ( PARTITION BY req_pos_id ORDER BY dist ASC )
                AS loc_timestamp
        FROM analysis_data
    )
```

The boggest part of the job is done right now: we have the points from the request, the matching, and the quality of matching. It's time to just perform updates on the table of waypoints.

Let's start from brand new waypoints:

```
    WITH
    -- request_data AS ( ... ) --> data from JSON request
    -- , session_data AS ( ... ) --> kernel from database
    -- , cross_data AS ( ... ) --> compute distances
    -- , analysis_data AS ( ... ) --> add match label and quality evaluation
    -- , classification_data AS ( ... ) --> final classification of the request

    , set_wps_new AS (
        SELECT
            'SARHL2_ID8651165355_DEVC' AS DEVICE_ID,
            'f783e7af1c6a5da5a2be77274dc204fe' AS SESSION_TOKEN_ID,
            '9237e74b12ccfd7b4c6cf5ea8c18c880' SESSION_TOKEN_INHERITED_ID,
            (max_session_id.max_id + ROW_NUMBER() OVER ()) AS LOCAL_POSITION_ID,
            ...
            FALSE AS ALIGNMENT_TYPE_FL,
            QUALITY_VL AS ALIGNMENT_QUALITY_VL,
            dist AS ALIGNMENT_DISTANCE_VL,
            ...
        FROM classification_data
        LEFT JOIN (
            SELECT
                MAX(pos_id) AS max_id
            FROM session_data
        ) AS max_session_id ON (1=1)
        WHERE NOT(WP_IS_REDUNDANT_FL)
    )
    , insert_new AS (
        INSERT INTO sar.F_HL2_STAGING_WAYPOINTS ( ... )
        SELECT
            ...
        FROM set_wps_new
        RETURNING *
    )
```

PostgreSQL allows to insert and return inserted rows from statement, which is useful for obtaining the final alignment table, as explained later. In this subquery, it is important to notice how new local IDs are generated. The subquery computes the max ID available before the insert of the new informations. SQL window function `ROW_NUMBER()` just enumerates rows, combined with a empty `PARTITION BY ... ORDER BY ...` clause, meaning that the database is free to choose any order for enumeration. Let's say we have 3 positions: `ROW_NUMBER()` will generate a column associating 1 to the first one, 2 to the second one, and 3 to the last one, in a not given order since there's no need here to assign a index given for instance a date. The new local position will be the sul of such a "row number" index plus the max ID.

For aligned points, the question is a bit more tricky, since maybe these points are already contained in the session of the client providing the informations; duplicates are possible adding already known local indexes, leading to anomalies and more complicated queries. To avoid such a situation, the query selects only the waypoints not yet recorded in the current client session by a simple subquery. Here's the SQL implementation:

```
    WITH
```

```
-- request_data AS ( ... ) --> data from JSON request
-- , session_data AS ( ... ) --> kernel from database
-- , cross_data AS ( ... ) --> compute distances
-- , analysis_data AS ( ... ) --> add match label and quality evaluation
-- , classification_data AS ( ... ) --> final classification of the request
-- , set_wps_new AS ( ... ) --> select of brand new points
-- , insert_new AS ( ... ) --> INSERT brand new points

, set_wps_aligned AS (
    SELECT
        ...
    FROM classification_data
    WHERE WP_IS_REDUNDANT_FL
    )
    , insert_history AS (
    INSERT INTO sar.F_HL2_STAGING_WAYPOINTS ( ... )
    SELECT
        ...
    FROM set_wps_aligned
    RETURNING *
    )
```

The alignment table is generated at the very end of the process, just taking a UNION of the two INSERT operations, and filtering on those records for which the local ID is different from request ID.

```
WITH
-- request_data AS ( ... ) --> data from JSON request
-- , session_data AS ( ... ) --> kernel from database
-- , cross_data AS ( ... ) --> compute distances
-- , analysis_data AS ( ... ) --> add match label and quality evaluation
-- , classification_data AS ( ... ) --> final classification of the request
-- , set_wps_new AS ( ... ) --> select of brand new points
-- , insert_new AS ( ... ) --> INSERT brand new points
-- , set_wps_aligned AS ( ... ) --> select matched points
-- , insert_history AS ( ... ) --> INSERT points not yet recorded in history

SELECT DISTINCT
    REQUEST_POSITION_ID,
    LOCAL_POSITION_ID AS ALIGNED_POSITION_ID
FROM set_wps_new
WHERE REQUEST_POSITION_ID <> LOCAL_POSITION_ID
UNION
SELECT DISTINCT
    REQUEST_POSITION_ID,
    LOCAL_POSITION_ID AS ALIGNED_POSITION_ID
FROM set_wps_aligned
WHERE REQUEST_POSITION_ID <> LOCAL_POSITION_ID
```

The result of the processing is a lookup table containing only the IDs of the waypoints which have been chanced coordinates due to the alignment, with the corresponding local ID starting from the request ID. It is reasy to be returned to the client, as well as to be leveraged later in the upload process.

A problem which have not been addressed in this transaction is the possibility to have different points which are near enough inside the request itself; for instance, it is not handled the case in which the input contains both (0,0,0) and (0,0.5,0) which are very close points.
To avoid this problem, a pre-filtering of the points inside the request is resquired. This situation could arise when the client and the server work on different set of parameters, and it does not affect if the points, which are very close inside the same request, are also near enough to a point inside the database.

8.4.3 Paths pre filtering

Along with the Upload request, the client provides not only the waypoints, but also the paths found for linking them.

Before starting with the real storage, a pre-filtering step is required. The paths processing is performed immediately after the waypoints processing, ans it could happen (i happened during the simulations) that sometimes two near points, linked with a regular path, are close enough to be aligned with the same point. This situation will produce a path starting from and ending to the same point, which is obviously wrong. To solve this anomalous case, it has been provided this mechanism:

1. alignment table from waypoints processing is mapped into a dictionary (this enables to process data with Python, since data are of type list of tuples when returned by the PostgreSQL connector)

2. for each path inside the request, find the referring aligned waypoint for each end of the path. Keep the path from the request only if the two ends are still different after processing.

This anomaly has been solved on the server side, but unfortunately not in HoloLens2 as explained later.

Another not handled case happend when two paths `(A, B)` and `(B, C)` are provided, but `A` is aligned with `C`, creating two paths satisfying tht rule but still redundant. Also this case can be corrected with a simple update of the Python script for pre-filtering, introducing a set of tuples and going to check if `(C, B)` is aligned when the system tries to insert the path `(B, C)`. Noticeable, nevertheless, that this situation is partially handled by the processing query, hereby explained.

Obviously there's a third irregualr situation, in which measurements contain a cycle: the case above is just a variation of the one of the explicit cycle. This situation should to be avoided, but, in terms of best effort, it is not required to intercept all the possible anomalies, especially at staging level. So, we could accept this kind of things at this point.

8.4.4 SQL Paths Processing

As done for the waypoints, the processing of paths is performed in only one query immediately after pre-filtering for improving performances: as said, database is more efficient than a simple (brute-force) Python processing.

The first step is to load paths from the request. The method is the same as before: PostgreSQL has a easy way to inject JSON code with a prepared statement directly into the query. It will be like this one:

```
CREATE TYPE json_schema AS (
    wp1 BIGINT,
    wp2 BIGINT,
    dist FLOAT,
    pt_timestamp TIMESTAMP
);

WITH
request_data AS (
SELECT DISTINCT
    wp1,
    wp2,
    dist,
    pt_timestamp
```

```
    FROM JSON_POPULATE_RECORDSET(NULL::json_schema,
    '[
--------{"wp1":-1,-"wp2":-0,-"dist":-0.0,-"pt_timestamp":-"2023/11/01-16:27:30"},
--------{"wp1":-2,-"wp2":-1,-"dist":-0.0,-"pt_timestamp":-"2023/11/01-16:27:31"},
--------...]')
    )
```

As base dataset of waypoint, here is not taken the kernel: the query uses the waypoints inside the same user session, perforing a selection on the basis of `SESSION_TOKEN_ID`.

```
    WITH
    -- request_data AS ( ... ) --> JSON paths from request

    , waypoints_base AS (
        SELECT *
        FROM sar.F_HL2_STAGING_WAYPOINTS
        WHERE SESSION_TOKEN_ID = 'f783e7...'
    )
```

At the time that snippet is launched into the database, its result will contian all the waypoints, included the ones from the previous step. The next step is to use this result to assign local IDs to the paths. Very important to point out that the JOIN operation is performed using the `REQUEST_POSITION_ID` directly written into the table. The objective is to assing primary keys and distance to the records from the request: the number of recors inside the result from this step will be the same of the paths the the system is trying to load.

```
    WITH
    -- request_data AS ( ... ) --> JSON paths from request
    -- , waypoints_base AS ( ... ) --> known waypoints in the client session

    , paths_renamed AS (
        SELECT
            wp1map.SESSION_TOKEN_ID AS SESSION_TOKEN_ID,
            wp1map.SESSION_TOKEN_INHERITED_ID AS SESSION_TOKEN_INHERITED_ID,
            wp1map.U_REFERENCE_POSITION_ID AS U_REFERENCE_POSITION_ID,
            wp1map.F_HL2_QUALITY_WAYPOINTS_PK AS WAYPOINT_1_STAGING_FK,
            wp2map.F_HL2_QUALITY_WAYPOINTS_PK AS WAYPOINT_2_STAGING_FK,
            dist(
                wp1map.UX_VL, wp1map.UY_VL, wp1map.UZ_VL,
                wp2map.UX_VL, wp2map.UY_VL, wp2map.UZ_VL
                ) AS PATH_DISTANCE,
            COALESCE(
                request_data.pt_timestamp,
                CURRENT_TIMESTAMP
                ) AS CREATED_TS
        FROM request_data
        LEFT JOIN waypoints_base
            AS wp1map
            ON( request_data.wp1 = wp1map.REQUEST_POSITION_ID )
        LEFT JOIN waypoints_base
            AS wp2map
            ON( request_data.wp2 = wp2map.REQUEST_POSITION_ID )
    )
```

This step can be deeply reviewed. First of all, as said, column `REQUEST_POSITION_ID` can be deleted since it is of no use from a practical point of view, except for working history if needed. In any case, it should not be used to let different processing steps to communicate: it is a poor choice, since this technical matter has to be addressed working within the same memory space of the transaction, and not at database level. It should have been implemented a way to leverage Python variables in order to not to need a column working in this way. A better JSON input, with a more structured information about alignment will solves the problem. It remains only the matter of restructuring the intermediate steps of the waypoints processing query, and it can be done mainly in two ways:

- review the output from waypoints processing in a way that allosws to extract missing information to attach later to the input paths later in the pre processing
- Postgres provides also methods to export data in JSON format; they could be useful to export multiple tables inside the same output

Another pboblem here is the concurrency. This table cannot write in the scope of the same transaction: PostgreSQL does not allow to SELECT updated informations from tables after INSERT when BEGIN TRANSATION is issued. *We need transaction!* Disabling them is simply unacceptable, since there could happens two writings at the same time, producing unpredictable results and leading to bugs which could be cumbersome to understand, especially in a collaborative application.

Anyway, the query right now can successfully transform data from raw input to a schema suitable for import operations. The next step is to select not redundant paths before insert, and the query is like this:

```
WITH
-- request_data AS ( ... ) --> JSON paths from request
-- , waypoints_base AS ( ... ) --> known waypoints in the client session
-- , paths_renamed AS ( ... ) --> enforce schema

, paths_renamed_filtered AS (
    SELECT
        *
    FROM paths_renamed
    WHERE ( WAYPOINT_1_STAGING_FK, WAYPOINT_2_STAGING_FK ) NOT IN (
        SELECT
            WAYPOINT_1_STAGING_FK, WAYPOINT_2_STAGING_FK
        FROM sar.F_HL2_STAGING_PATHS
        WHERE 1=1
        AND SESSION_TOKEN_ID = 'f783e7af1c6a5da5a2be77274dc204fe'
        UNION ALL
        SELECT
            WAYPOINT_2_STAGING_FK, WAYPOINT_1_STAGING_FK
        FROM sar.F_HL2_STAGING_PATHS
        WHERE 1=1
        AND SESSION_TOKEN_ID = 'f783e7af1c6a5da5a2be77274dc204fe'
    )
)
```

For both the cases, paths are taken from the known paths. Important to say that, despite the waypoints are updated, paths are not updated, hence this process finds only the paths known at the time immediately before the upload operation.

The next step is just to insert paths into the paths table. Paths are attached to the current client session, and become known informations for the server, as they are for the device.

References

Agrawal, R., Faujdar, N., Romero, C.A.T., Sharma, O., Abdulsahib, G.M., Khalaf, O.I., Mansoor, R.F. & Ghoneim, O.A. (2023). Classification and comparison of ad hoc networks: A review. *Egyptian Informatics Journal*, **24**, 1–25. 3, 7

Al-Shareeda, M.A. & Manickam, S. (2022). Man-in-the-middle attacks in mobile ad hoc networks (manets): Analysis and evaluation. *Symmetry*, **14**. 3

Alshamrani, A. & Bahattab, A. (2015). A comparison between three sdlc models waterfall model, spiral model, and incremental/iterative model. *International Journal of Computer Science Issues (IJCSI)*, **12**, 106. 11

Anjum, S.S., Noor, R.M. & Anisi, M.H. (2015). Survey on manet based communication scenarios for search and rescue operations. In *2015 5th International Conference on IT Convergence and Security (ICITCS)*, 1–5. 3, 7

Aschenbruck, N., Gerhards-Padilla, E., Gerharz, M., Frank, M. & Martini, P. (2007). Modelling mobility in disaster area scenarios. In *Proceedings of the 10th ACM Symposium on Modeling, Analysis, and Simulation of Wireless and Mobile Systems*, MSWiM '07, 4–12, Association for Computing Machinery, New York, NY, USA. 3, 4, 7

Cao, S., Sun, Z., Diwondi Hermine Gaelle, S. & Su, G. (2021). Design of a portable life search and rescue platform for complex disaster. In *2021 International Conference on Public Management and Intelligent Society (PMIS)*, 363–366. 3, 4, 5

Cook, D.J., Augusto, J.C. & Jakkula, V.R. (2009). Ambient intelligence: Technologies, applications, and opportunities. *Pervasive and Mobile Computing*, **5**, 277–298. 3

Dioniso (2023). Progetto dioniso, tecnologie innovative di domotica sismica per la sicurezza di edifici ed impianti. `https://dioniso.dibris.unige.it/`. 1

Docker (2023). Docker overview. `https://docs.docker.com/get-started/overview/`. 123

Drew, D.S. (2021). Multi-agent systems for search and rescue applications. *Current Robotics Reports*, **2**, 189–200. 3

Eltahlawy, A.M., Aslan, H.K., Abdallah, E.G., Elsayed, M.S., Jurcut, A.D. & Azer, M.A. (2023). A survey on parameters affecting manet performance. *Electronics*, **12**.

FastAPI (2023). Fastapi overview. `https://fastapi.tiangolo.com/`. 123

Grassi, L., Ciranni, M., Baglietto, P., Recchiuto, C.T., Maresca, M. & Sgorbissa, A. (2023). Emergency management through information crowdsourcing. *Information Processing & Management*, **60**, 103386. 3, 5

Huang, X., Yang, J., Jin, W., Fang, Y. & Cui, L. (2021). Design and application of a practical decision-making and commanding platform for emergency rescue. In *2021 IEEE 12th International Conference on Software Engineering and Service Science (ICSESS)*, 105–108. 3

Jia, G., Li, X., Zhang, D., Xu, W., Lv, H., Shi, Y. & Cai, M. (2022). Visual-slam classical framework and key techniques: A review. *Sensors*, **22**. 14

Kandris, D., Nakas, C., Vomvas, D. & Koulouras, G. (2020). Applications of wireless sensor networks: An up-to-date survey. *Applied System Innovation*, **3**. 3, 7

Korir, F. & Cheruiyot, W. (2022). A survey on security challenges in the current manet routing protocols. *Global Journal of Engineering and Technology Advances*, **12**, 078–091. 3

Lavie, T. & Meyer, J. (2010). Benefits and costs of adaptive user interfaces. *International Journal of Human-Computer Studies*, **68**, 508–524, measuring the Impact of Personalization and Recommendation on User Behaviour. 3

Luksas, J., Quinn, K., Gabbard, J.L., Hasan, M., He, J., Surana, N., Tabbarah, M. & Teckchandani, N.K. (2022). Search and rescue ar visualization environment (save): Designing an ar application for use with search and rescue personnel. In *2022 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*, 488–492. 3, 8

Maceda, L.L., Llovido, J.L. & Serrano, J.E. (2018). System design for disaster risk damage assessment. In *2018 International Symposium on Computer, Consumer and Control (IS3C)*, 246–249. 3, 5

Microsoft (2023a). Azure data factory documentation. `https://learn.microsoft.com/en-us/azure/data-factory/introduction`. 6

Microsoft (2023b). Azure databricks documentation. `https://learn.microsoft.com/en-us/azure/databricks/`. 6

Microsoft (2023c). Azure spatial anchors documentation. https://learn.microsoft.com/en-us/azure/spatial-anchors/overview. 6

Microsoft (2023d). Azure storage account documentation. https://learn.microsoft.com/en-us/azure/storage/common/storage-account-overview. 6

Microsoft (2023e). Hololens2 solvers. https://learn.microsoft.com/en-us/windows/mixed-reality/mrtk-unity/mrtk2/features/ux-building-blocks/solvers/solver?view=mrtkunity-2022-05. 24

Microsoft (2023f). Hololens2 technical specifications. https://www.microsoft.com/en-us/hololens/hardware. 3, 14

Moudni, H., Er-Rouidi, M., Mouncif, H. & El Hadadi, B. (2016). Attacks against aodv routing protocol in mobile ad-hoc networks. In *2016 13th International Conference on Computer Graphics, Imaging and Visualization (CGiV)*, 385–389. 3

NASA (2023). Nasa earth facts sheet. https://nssdc.gsfc.nasa.gov/planetary/factsheet/earthfact.html. 30

Nginx (2023). Nginx overview. http://nginx.org/en/docs/beginners_guide.html. 123

Nunavath, V., Prinz, A. & Comes, T. (2016). Identifying first responders information needs: Supporting search and rescue operations for fire emergency response. *International Journal of Information Systems for Crisis Response and Management*, **8**, 25–46. 3, 4, 5, 8

Onwuka, L., Folaponmile, A. & Ahmed, M. (2012). Manet: A reliable network in disaster areas. *Journal Of Research In National Development Transcampus*, **9**, 1596–8308. 3, 7

Palumbo, A. (2022). Microsoft hololens 2 in medical and healthcare context: State of the art and future prospects. *Sensors*, **22**. 3

PgVector (2023). Pgvector on github readme. https://github.com/pgvector/pgvector. 123

PostgreSQL (2023). Postgresql overview. https://www.postgresql.org/about/. 123

Ramphull, D., Mungur, A., Armoogum, S. & Pudaruth, S. (2021). A review of mobile ad hoc network (manet) protocols and their applications. In *2021 5th International Conference on Intelligent Computing and Control Systems (ICICCS)*, 204–211. 3

Recchiuto, C.T., Scalmato, A. & Sgorbissa, A. (2017). A dataset for human localization and mapping with wearable sensors. *Robotics and Autonomous Systems*, **97**, 136–143. 1, 3, 6

REFERENCES

Reina, D.G., Toral, S., Barrero, F., Bessis, N. & Asimakopoulou, E. (2013). Modelling and assessing ad hoc networks in disaster scenarios. *Journal of Ambient Intelligence and Humanized Computing*, **4**, 571–579. 3, 7

Semiconductor, N. (2023). Thingy:91 technical specifications. `https://iotcreators.com/wp-content/uploads/2018/05/Nordic-Thingy-91-PB-1.2.pdf`. 10, 104

Seneviratne, S., Hu, Y., Nguyen, T., Lan, G., Khalifa, S., Thilakarathna, K., Hassan, M. & Seneviratne, A. (2017). A survey of wearable devices and challenges. *IEEE Communications Surveys & Tutorials*, **19**, 2573–2620. 3

Testa, F. (2022). *DIONISO - Wearable Sensors and Augmented Reality for Automous Mapping and Navigation in SearchRescue*. Master's thesis, University of Genova, project Available at `https://github.com/FraTesta/MySceneUnderstanding`. 1, 2, 3, 5, 6, 8, 99, 108

Van Westen, C. (2000). Remote sensing for natural disaster management. *International archives of photogrammetry and remote sensing*, **33**, 1609–1617. 3

Waharte, S. & Trigoni, N. (2010). Supporting search and rescue operations with uavs. In *2010 International Conference on Emerging Security Technologies*, 142–147.

Wang, R., Lu, H., Xiao, J., Li, Y. & Qiu, Q. (2018). The design of an augmented reality system for urban search and rescue. In *2018 IEEE International Conference on Intelligence and Safety for Robotics (ISR)*, 267–272. 3

Weinmann, M., Wursthorn, S., Weinmann, M. & Hübner, P. (2021). Efficient 3d mapping and modelling of indoor scenes with the microsoft hololens: A survey. *PFG–Journal of Photogrammetry, Remote Sensing and Geoinformation Science*, **89**, 319–333.

Williams, A., Sebastian, B. & Ben-Tzvi, P. (2019). Review and analysis of search, extraction, evacuation, and medical field treatment robots. *Journal of Intelligent & Robotic Systems*, **96**, 401–418. 3