POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering

Master's Degree Thesis

Mixed Real-Time Visualization Framework for FGB IoT sensors

Supervisors
Prof. Bartolomeo Montrucchio
Prof. Andrea Sanna

Candidate
Maria Giulia Canu
237023

July 2019

Written in LaTeX on July 11, 2019

Contents

List of Figures VI

List of Tables VIII

Listings IX

Abstract X

Acknowledgements XII

1 Introduction 1
- 1.1 General Description . . . 1
- 1.2 Fiber BRAGG sensor . . . 2
- 1.3 Document structure . . . 3

2 Previous work 4
- 2.1 System overview . . . 4
  - 2.1.1 Physical system . . . 5
  - 2.1.2 Interrogator . . . 5
  - 2.1.3 Middleware . . . 6
  - 2.1.4 Cloud Network . . . 6
  - 2.1.5 Analysis Framework . . . 7

3 State of art 8
- 3.1 Augmented Reality in Industry 4.0 . . . 8
  - 3.1.1 IoT Integration . . . 9
  - 3.1.2 Maintenance and Monitoring system . . . 11
- 3.2 Example of non AR monitoring system . . . 14

4 Proposed Real-Time Framework 16
- 4.1 General description . . . 16
- 4.2 Unity3D engine . . . 16
- 4.3 Data visualization . . . 18
  - 4.3.1 Shader-base Heatmap . . . 19
- 4.4 Data collection . . . 21
  - 4.4.1 Mongo DB . . . 22
  - 4.4.2 TCP-Based Protocol . . . 22

4.5 Available AR technologies . . . 24
4.5.1 Microsoft Hololens . . . 25
4.5.2 Vuforia . . . 26

5 Implementation Details 28
5.1 Common features . . . 28
5.1.1 Class structure . . . 29
5.1.2 Multi-thread communication . . . 37
5.1.3 Heatmap Shader . . . 38
5.1.4 Middleware connection . . . 40
5.2 Desktop solution . . . 41
5.2.1 Overview . . . 42
5.2.2 In game features . . . 43
5.2.3 Real time graph . . . 51
5.2.4 End simulation features . . . 52
5.3 Hololens solution . . . 54
5.3.1 MRTK v2 Release Candidate 1 . . . 54
5.3.2 Limitation . . . 55
5.3.3 Overview . . . 55
5.3.4 Marker-base recognition . . . 58
5.3.5 Sensors customization . . . 58

6 Test and Result 61
6.1 Measurement Log . . . 61
6.2 Performance: Case studies . . . 61
6.2.1 Case study: Carbon Fiber Reinforced Polymer sheet . . . 62
6.2.2 Case study: Fuselage ICARUS . . . 62
6.2.3 Case study: Emulator . . . 64
6.2.4 Case studies analysis . . . 65
6.3 Review and usability test . . . 66

7 Future Work 68
7.1 Missing features . . . 68
7.2 Custom image-tracking and real-time graph implementation . . . 68
7.3 Hololens 2 porting . . . 69

8 Conclusion 70

Acronyms 71

Bibliography 74

List of Figures

1.1 *Microsoft© Hololens* headset 2
1.2 FBG sensor structure 2

2.1 Framework architecture 5
2.2 *SmartScan©* interrogator 6

3.1 Augmented Reality Examples 8
3.2 Example of discrepancy check in AR 10
3.3 Example of usage IoT in real-time machine inspection 11
3.4 AR Wireless Power Measurement System for a Manifacturing Line . . 12
3.5 Pipe planning application on Android tablet 12
3.6 *BMW©* 2015 AR application 13
3.7 AR application for production quality 13
3.8 FOV comparison 14
3.8a FOV comparison between AR HMD 14
3.8b FOV comparison between AR and VR headset and human one 14
3.9 FBG bi-state monitoring system 14

4.1 Unity Logo 17
4.2 Example of rendering pipeline 20
4.3 Example of the visualization system 21
4.4 Steps to execute the Air-Tap gesture 26
4.5 Steps to execute the Bloom gesture 26
4.6 Vuforia Engine examples 27
4.6a Image Target Example 27
4.6b Model Target Example 27

5.1 Representation of the sensors in the framework 29
5.1a Temperature sensor model 29
5.1b Displacement sensor model 29
5.2 Main framework's GUI elements 34
5.3 Multi-thread connection schema for *MongoDB* 38
5.4 Multi-thread connection schema for TCP Connection 38
5.5 Multi-thread connection schema on middleware client 41
5.6 Desktop application home view 42
5.7 Parameters to customize during the `import` status 45

5.8 Custom HeatMap colors GUI . . . 47
5.9 Update heatmap on the model and UI . . . 47
5.10 `ChangePos` status main screen . . . 48
5.11 Selected sensor's glowing effect . . . 49
5.12 Example of the real-time graph with 64 sensors . . . 51
5.13 Example of a Log file with five active sensors . . . 53
5.14 Example of line graph generated by the python's script . . . 53
5.15 MRTK logo . . . 54
5.16 *Hololens* main GUI . . . 56
5.17 Hololens's virtual keyboard . . . 57
5.18 *Vuforia*'s main scene elements on the Unity scene . . . 58
5.19 *Menu sensor* GUI for sensor's insertion . . . 59
5.20 *Menu sensor* GUI for displacement sensor's customization . . . 60
5.21 `Monitoring` phase example on the ICARUS wing . . . 60

6.1 Comparisons between line graph obtained from the sample received from the interrogator . . . 62
6.1a Visualization framework real-time line graph . . . 62
6.1b *Matlab* line graph . . . 62
6.2 Fuselage displacement *Hololens* test . . . 63
6.3 *ICARUS*'s first wing test . . . 63
6.4 Comparisons between line graph obtained from the sample received from the interrogator . . . 64
6.4a Visualization framework real-time line graph . . . 64
6.4b *Matlab* line graph . . . 64
6.5 Visualization Framework Delay overview . . . 66

List of Tables

4.1 ConfigPacket payload description 24

6.1 CFRP sheet measured delay in ms 62
6.2 Desktop: Fuselage displacement measured delay in ms 63
6.3 *Hololens*: Fuselage displacement measured delay in ms 63
6.4 Desktop: Emulator delay in ms 64
6.5 *Hololens*: Emulator delay in ms 65
6.6 Desktop: performance with and without the real-time graph 65

Listings

5.1 Status enum description 28
5.2 `MongoDBManager` class 30
5.3 `TCPManager` class 31
5.4 `MonitoredObject` class 33
5.5 `GUIManager` class 35
5.6 `GameManager` class 36
5.7 `Sensor` struct 37
5.8 vertOutput struct description 39
5.9 Fragment shader original code 39
5.10 Fragment shader modified code 40
5.11 `GUIManager` import methods 44
5.12 Calculation of sensor's position 45
5.13 `MonitoredObject` import methods 46
5.14 Selected sensor's model position update 49
5.15 Selected sensor's model rotation update 50
5.16 `GUIManager` graph methods 52
5.17 `NetworkInterface` difference between desktop and Hololens implementation 57

Abstract

The objective of this thesis is to design one of the first real-time visualization frameworks for Fibre Bragg Grating (FBG) sensors, which supports both VR (virtual reality) and AR(Augmented reality) technology. Those sensors measure the variation of the refractive index of the optical fibre core caused by strain or temperature. Nowadays, this technology had spread out in a different field, usually as part of a preventive monitoring system, because of its advantage to work in environments subject to EMI and strong electrical potential where other sensors cannot be used. At the same time, technologies such as Virtual Reality and Augmented reality are widely used in the emerging [4], aerospatial, automotive [6] and industrial plant [14], include AR devices in their production line for decades. Indeed, it is one of the best technology for maintenance and quality control tasks because of the mix of virtual visual and textual information superimposed as a hologram on the physical object. Another exploring branch for AR is *Internet of Things* (IoT), with systems commonly known as ARIoT. Those tools are synchronized with a cloud platform that collects data from different sensors and shows in real-time the whole set of details needed by technicians to solve their task. Workers discern more easily how the object is working and how to react in case of failures. They show more concentration on doing their task and a significant increase in their performance in terms of speed and problem solving (if compared to equivalent desktop applications). The best results are seen with Head Mounted Display instead of Hand Handled Display (smart-phones or tables) because the operator's hands are free and there is no necessity to shift between instructions and the devices that need maintenance.

The proposed work is an improvement of a previously developed framework created as a general open source tool for any FBG-based system. Its integration is justified by the lack of real-time monitoring systems for those sensors in literature.
Two different applications were developed with Unity3D engine, one for desktop system and another one for the *Microsoft© Hololens* headset. Both accept any kind of 3D models as input and allows to fully customize the position and rotation of the sensors and their size on its surface. They communicate with a middleware client that directly retrieves data from the sensors. This choice was necessary to reduce the response time of the visualization framework now able to display data at a satisfactory rate. To increase the performance even more both application are multithread, the main thread manages the GUI and all the user inputs and the secondary thread manages the socket. A dispatcher class permits the communication between secondaries and the main thread.

The desktop application is provided with a line graph that is updated in real time to show the trend of the sensor during the monitoring phase. Furthermore, it permits to change model directly in the application without the need to recompile the program in the Unity Editor. The AR application superimposes the hologram of the monitored object in the correct location and orientation by exploiting the *Vuforia Engine* image-target recognition system.
The project was developed in collaboration with the DAUIN, DIMEAS and DISAT department of Politecnico di Torino, as part of the Inter-Departmental Center for Photonic technologies (*PhotoNext*). The case study was performed with two main physical systems: a Carbon Fiber Reinforced Polymer sheet (CFRP) and the fuselage of the *ICARUS* Unmanned Aerial Vehicle developed by DIMEAS. In both cases, the objective was to monitor the displacement of the two systems in real time. They represent the intensity of the retrieved value with a different colour based on the difference with idle value.
The application was used by some members of the *ICARUS* team and showed fast response time and to be easy and intuitive to use. In particular, the *Hololens* implementation was the one that had the biggest impression, so much that it was suggested to use it in future flight tests.

Acknowledgements

Thanks to Professor Bartolomeo Montrucchio and Professor Andrea Sanna for the opportunity they give me with this thesis.

Thanks to Mauro, Edoardo and Mohammad, for all the support and help during this months.

Thanks to my family and friends to support and comfort me during this month of huge stress.

Last but not least, thanks to Alessandro that is always on my side and encourage me to do my best even if we are very far from each other.

Chapter 1

Introduction

The following chapter presents a general description of the proposed framework plus a brief description of FBG sensor.

1.1 General Description

This thesis aims to implement a mixed real-time visualization system. It is an improvement of a previously proposed IoT architecture for a generic system that uses FBG-base sensors [5]. The whole architecture is required as part of the *PhotoNext* group research project regarding FBG technology and system monitoring [17].

FBG sensor is an optical device able to measure temperature, strain, and displacement of a generic physical system. Nowadays it is widely used as a preventive monitoring tool in different fields thanks to its advantage to work in extreme condition, such as explosive and electronic radiation environment, or situations with a substantial electromagnetic interference or potential difference. The only way to retrieve data from these sensors is with a particular piece of hardware called interrogator. It extracts both raw and peak data and forwards the samples to ad-hoc software, usually developed by the same vendor of the interrogator.
Other than this software, no other open source real-time visualization system seems to exists for those type of sensors. In addition, no prototype was found that shows in real time the location of a FBG sensor upon a 3D model and its value in literature.

During the last 20 year, solutions that integrate VR/AR become more and more widespread and advanced. Monitoring and maintenance system made AR one of the most adapted and productive tools in the different industrial field, with its peak in the automotive [6, 16] and aerospatial one. Its feature to superimpose mixed type of visual information on the physical world increases the quality of an operator's work. Especially headset or wearable devices deliver better performance and higher concentration during work [4]. Because of its widespread use as a maintenance tool, AR devices are usually connected to IoT networks [21]. Detailed information, update in real-time, are displayed on the screen of those devices without the need of other computer interfaces.
Based on those observations, a real-time monitoring framework for FBG IoT sensors was developed.

It is split into two application: a VR application for desktop/laptop and an AR application for the *Microsoft© Hololens* headset.

The VR application is proposed in situations where the user is far from the monitored object or if the object itself is in motion. It shows a CAD model with information about the location of the sensors upon its surface, based on the measurement from the sensor the area around it changes colour. Besides, it displays a message box with the id of the sensor, and its last measurement retrieve from the database.

Figure 1.1: *Microsoft© Hololens* headset

Source: [13]

The AR application, instead, is meant to in-site monitoring where the system has small-medium dimensions. In this case, the CAD model is superimposed on the monitored object by using a mark based system. The user is free to move around it and interact with it using hand gestures.

1.2 Fiber BRAGG sensor

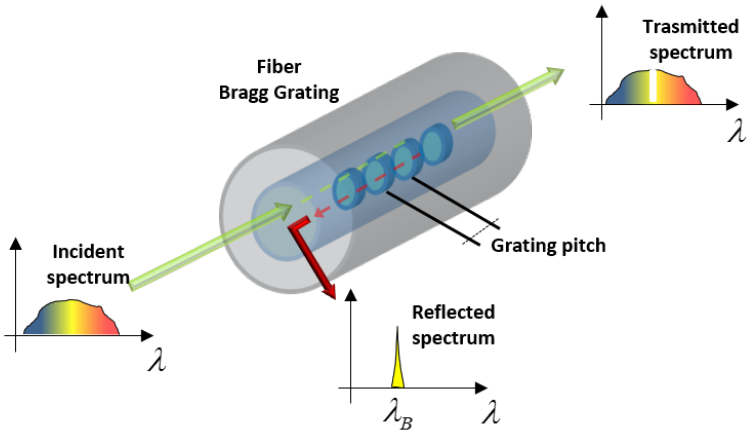


Figure 1.2: FBG sensor structure

Source: [27]

An FBG sensor can be defined as a *"very high sensitive and versatile optical device for measuring several physical parameters including for example: strain, temperature, pressure, vibration and displacement"* [27]. In its basic form is build as *"a permanent periodic refractive index modulation inscribed in the optical fiber core exploiting photosensitivity"* [27]. FBG sensors *"exploit the presence of a resonance condition for which they reflect incident light at the so-called Bragg wavelength defined as:"* [27]

$$\lambda_B = 2 \cdot n_{eff}\Lambda \tag{1.1}$$

"Where n_{eff} is the effective core refractive index of the fundamental mode and Λ is the grating pitch." [27]
Nowadays, it is possible to define different sensor's profile based on parameters such as peak reflectively and wavelength bandwidth. The possibility to perform this customization during the production stages permits to reduce costs for mass production and also to be adopted in large structures which require a huge number of sensors. Any change in the refractive index or sensor pitch caused by external influences such as temperature or strains manifests as a wavelength shift, according to the following formula:

$$\Delta\lambda_B = \lambda_B[(\alpha + \zeta)\Delta T + (1 - p_\varepsilon)\Delta\varepsilon] \tag{1.2}$$

"[...] α and ζ are the thermal expansion and thermo-optic coefficients, p_ε is the effective photo-elastic constant of the fiber material and ΔT, $\Delta\varepsilon$ are the applied temperature and longitudinal strain variations". [27] General Bragg wavelength value is approximately 1550 μm. Thanks to the fact that the parameter to be measured are part of the wavelength value and optical fibers feature the low attenuation and transmission capacity, FBG sensors permit a reliable signal detection even if the sensor and control unit far forms each other.
Thanks to the usage of multiplexing techniques, it is possible to manage multiple sensors on the same cable. These are divided into two major categories:

- *Wavelength Division Multiplexing (WDM)* where multiple FBG sensors can be cascaded in a single optical link with different nominal center wavelengths.
- *Time Division Multiplexing (TDM)* where the sensors have the same central wavelength, but they can be identified using the signal flying time.

1.3 Document structure

This document is divided in seven chapters: a brief summary of the previous work (chapter 2), a general description of the state of the art of Augmented Reality (AR) in *Industry 4.0* (chapter 3), a in-detail description proposed real-time framework (chapter 4), its implementation details (chapter 5), test and results of the developed tool (chapter 6), possible improvements (chapter 7) and the conclusion (chapter 8).

Chapter 2

Previous work

This chapter summarizes the previous work by giving a general description of the framework and then a more detailed one for each layer. An in-depth report can be found on [5].

2.1 System overview

The aim of this thesis is to expand a previously designed software architecture for a generic system which integrates a set of FBG sensors independently from the physical object or devices used. It was requested by the *PhotoNext* research group project that focuses on FBG sensors and their monitoring system.

The architecture allows to integrate FBG sensors into a general IoT platform easily, but in the future it is planned to be extended to any type of sensors.
It features a middleware layer which collects data and forwards it to a low latency cloud database. The proposed implementation is fully open source and uses popular and well tested third-party software, also open source.

It is subdivided into five main layers:

- **Physical system**: the monitored object which uses FBG sensors.
- **Interrogator**: the device used as interface for the sensors.
- **Middleware**: layer which provides connectivity to the interrogator.
- **Cloud network**: it stores and collects data in a safe and reliable way
- **Visualization and Analysis framework**: it allows to observe the measurements in real time by using the VR/AR framework or to analyze data after the measurement.

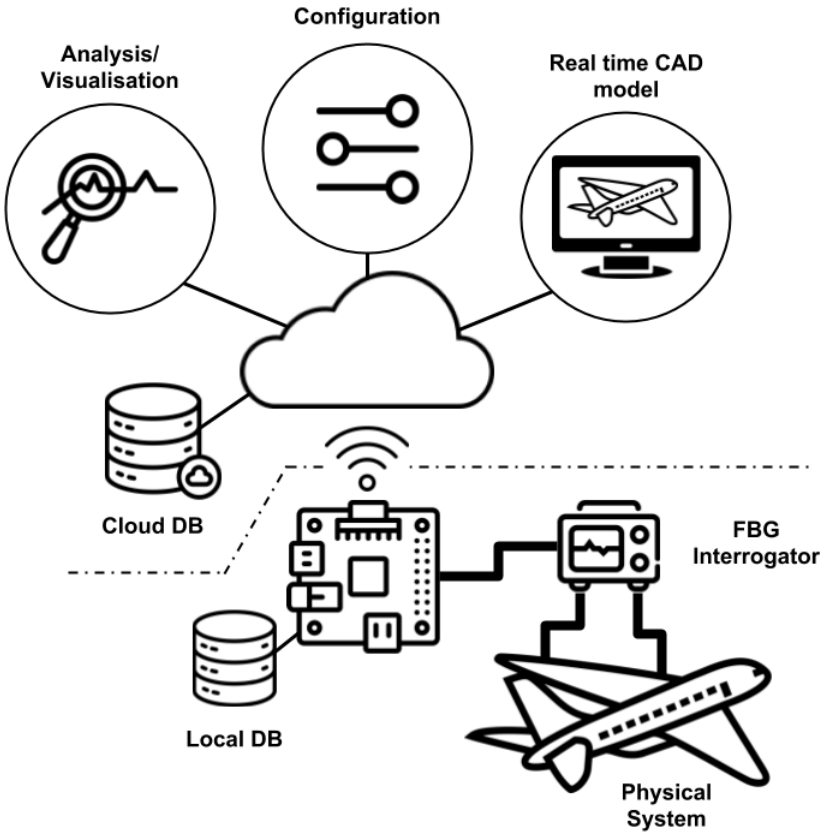


Figure 2.1: Framework architecture

Source: [5]

2.1.1 Physical system

The physical system layer consists of a general object monitored by a set of FBG sensors. It could be any system that requires continuous temperature (or displacement) checks in specific points of its structures. The reasons could be various, starting from identify structural issues in real-time or collecting data for offline analysis.

2.1.2 Interrogator

The interrogator is a hardware device able to read data from a set of FBG sensors. It provides both raw and peak data of the sensors; the last is obtained by a particular algorithm from the raw data. Usually, any interrogator vendor provides software able to read data from the fibre. The one used as a case study was the *SmartScan*© one from *SmartFibres*©. It communicates with the *SmartSoft Application Software*© using a custom UDP protocol on a LAN connection. It supports up to 64 FBG (4 channels, 16 gratings per channel).

Figure 2.2: *SmartScan*© interrogator

Source: `https://www.smartfibres.com/products/smartscan`

2.1.3 Middleware

The middleware is the central piece of the architecture which provides the IoT connectivity between the interrogator and the cloud network. It was implemented a C++ abstract interrogator class to keep the framework detached from the physical interrogator. The provided middleware client is optimized to work in a multi-thread environment able to handle continuous raw and peak data at high speed. The received packet needs to be parsed in a new format compatible with the one supported by the cloud platform database.
Each measurement is associated with different metadata (position of the sensors on the object, time-stamp, variation, etc.). Metatada is crucial for the fact that data inside the cloud is used by different layers of the architecture. Each one has a different purpose, such as real-time visualization for the AR/VR framework and offline analysis for the analysis one.

2.1.4 Cloud Network

The cloud network layer is implemented using the open source *KaaIoT* platform, a framework commonly used in the IoT application. It supports a wide range of non-relational databases, more efficient in term of reading/writing speed compared to the relational ones. *KaaIoT* improves its performance even more by allowing multiple

insert operations on different databases at the same time. Thus, the system should be fast enough to collect the data from the interrogator and store it securely. It provides all the security measures by using the TLS protocol for the data packet and a token-based system to authenticate third-party applications. *KaaIoT* supports data in a particular format specified by a JSON schema, and it permits to fully customize the data inside the structure depending on the aim of the application. In this particular case, only two types of schemas are used, one for raw data and another for peak data.
The NoSQL database used in the proposed model is *MongoDB*; it was chosen for its high-speed performance for insertion and retrieve operation, as shown in [3].

2.1.5 Analysis Framework

The analysis framework is the highest layer of the architecture in terms of abstraction, and it provides an offline analysis of the received data by providing different type of graphs.
The objective was reached by using the web-based notebook *Apache Zeppelin*, fully integrated with the *KaaIoT* platform because it is *Apache* based. It was proposed only for evaluation purpose and has not to be considered as a final element for the architecture. *Apache Zeppelin* requires a *MongoDB* interpreter to retrieve the JSON objects correctly.

Chapter 3

State of art

The following chapter presents a general overview of the State of The Art of the Augmented Reality usage in industry. It begins with a definition of *Industry 4.0*, followed by a description of what ARIoT is and it concludes by some implementation example in the monitoring and managing field.

3.1 Augmented Reality in Industry 4.0

The term *Industry 4.0* was introduced for the first time by the German government in 2013; it was a strategic initiative aimed at the digitalization of factories for achieving higher productivity, efficiency, and flexibility.
Today it is commonly referred to as the fourth industrial revolution for its integration into the production line new technologies such as IoT, cloud computing and cognitive computing (voice recognition, computer vision, machine learning, etc.) [19].

Figure 3.1: Augmented Reality Examples

Source: [21]

Augmented Reality (AR) stands out as a mix of technologies inside the cognitive

computing category. AR is a technology aimed to "enhance" the real world perception with virtual objects that are superimposed on a physical object. The first example of this technology can be dated 1968 when Ivan Sutherland performed its first experiment creating the first Head Mounted Display, the first device to support both AR and VR. The system, called "the Sword of Damocles" was only able to display a wired cube in the middle of the user field of view (FOV) and be able to move around it.
In the following years, it was proposed the first documented implementation of Augmented Reality, for the industrial purpose, was developed by the two Boeing engineers (Tom Caudell and David Mizell) in 1990 which developed a head-mounted display to visualize a plane's specific schematics on a board [1].
Since there, researchers in different universities delved into this promising technology and only after few years *ARTookit* was exposed for the first time at SIGGRAPH in 1999. With its open source release in 2001, the usage of AR spread out in different industrial fields, especially nowadays because of the reduced costs of the devices able to implement it [15]. Mobile devices like smartphones and tablets now feature a full range of sensors and have the hardware capability to execute an Augmented Reality application with good performance. Cheaper Head Mounted Displays (HMD) and other wearable devices are released into the market and, with it, a large amount of open source SDK, which increases the number of companies able to invest in it.

The domain of application of IAR is vast, as described in [1, 15] it is a powerful tool for helping workers during their task and can easily teach them new procedures with intuitive step by step applications. It speeds up quality control and maintenance operations, reduces the cause of errors and increases efficiency with its usage of visual, textual, and auditorium information. IAR decreases the cost even during the early design and manufacturing stages, in this way the number of physical prototypes shrinks because a mixed reality environment allows a virtual model to appear as a hologram into a real environment.
For this reason, it is also essential to check the differences between the mock-up and the CAD model. The model has to match perfectly the virtual one to progress into the planning and finally the construction of the product. Example of this application was seen since 2007 as described in [34]. The operation was usually done not in real-time because of the high computational load, but nowadays the process of "discrepancy check" is almost immediately, one implementation of this is reported in [33].

3.1.1 IoT Integration

Internet of Things (IoT), also called to as "Internet of Everything", defines a network of devices capable of Internet connectivity that communicates with each other and with external systems. It is one of the pillars of *Industry 4.0* and has a vast range of application that goes from house automation and energy management to transportation and medical assistant [2, 5, 21, 35]. Because the inclusion of new element in the network is straightforward, it is no surprise that applications which mix both IoT and AR already exist.

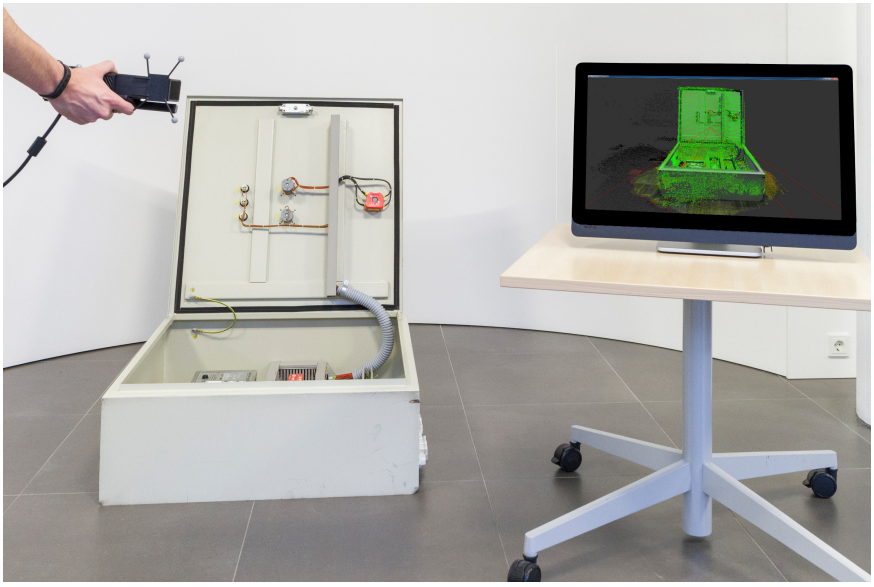


Figure 3.2: Example of discrepancy check in AR

Source: [33]

In a monitoring environment, IoT systems collect a huge amount of information that needs to be visualized in real-time. Data is usually saved in massive databases and operators should retrieve them in a fast and efficient way. By using an AR app on a smartphone (or tablet) connected to the system, it possible to show the details of a component as text or images upon the real object immediately. The usage of Head Mounted Display (HMD, headset with smartglasses) produces even better results, hands are free and any maintenance operation can be executed more efficiently without any distraction. In fact, the use of Hand-Held Display (HHD) can reduce the worker's attention because it has to switch continuously between instructions and the industrial process itself as described in [4]. In general, Augmented Reality and IoT (ARIoT) is used in monitoring, fault diagnostic and inspection application.

Example of this technology are described in [21] and is clear that prototypes of ARIoT can be found in different industrial fields. One of them was developed for helping farmers identifying insects species. Not all farmers are not able to recognize which ones are beneficial or not for the growth of the crops, so the usage of AR can simplify the task. Another model represents the underground networks by exploiting the usage of AR, GNSS, GIS and a set of sensors. In this way, its possible to avoid damaging existing infrastructure during digging and prevent a huge money loss. Another method used for ARIoT was proposed for providing manuals for different industrial equipment. It is able not only to execute a target recognition but also to display the necessary details upon the observed object. Nestor Lobo et al. implemented the *Intelli-Mirror* model used to detect a user and display upon it images of clothes. It also displays information about the garment such as name price, size, and name.

Figure 3.3: Example of usage IoT in real-time machine inspection

Source: [35]

3.1.2 Maintenance and Monitoring system

"'Maintenance' means all the corresponding technical and administrative actions, including monitoring and control operations, intended to maintain [...] an entity in a specified state or under conditions Dependability data, (Availability, Reliability, Maintainability and Security) enabling it to perform a required function" [7]. Any company highly invests in maintenance to reduce costs, time and increase the quality of their products. Maintenance tasks are performed periodically to check the status of a device, but, if the next control is performed too far in time, it is hard to know the exact moment in which an object breaks. Preventive systems are designed for this scope. The usage of programmable logic controllers, microcomputers, etc. offers a system that collects the status of the monitored devices in real-time. With AR is possible to visualize those data immediately and to apply any needed procedures to solve the fault or prevent it.

The number of projects and prototypes, which integrate AR in maintenance procedures, increases every year. As described in [7], a lot of models exploit mobile Augmented Reality and the usage of hand gesture to facilitate the execution of a task. Also, there are examples of collaborative systems where different technicians share the same mixed reality view and work together at the same procedure. The same [7], proposed a preventive maintenance system for a cement company. The application is developed with *Unity3D*© and *ARToolKit* for android devices and is able to show in real-time and on-site all the data related to any pieces of equipment.

Amici et al. [1] proposed a measurement system for a manufacturing line of an office kit in 2018. Each part of the line is equipped with a QR code to display information about voltage, current, power and energy consumption. Besides, the system is programmed to send an alarm in case of over-voltage or over-current.

Palmarini et al. [16] summarized the AR applications in maintenance by using an SLR approach, in other works they search, appraise, synthesize and analyze all the paper available at the moment. The document shows a higher interest in AR usage in

aviation, automotive, and industrial plant. Its introduction produces a reduction in terms of human error and time performing the procedure. Even if with good results, the technology does not seem to be mature in terms of robustness and reliability.

Figure 3.4: AR Wireless Power Measurement System for a Manifacturing Line

Source: [1]

Figure 3.5: Pipe planning application on Android tablet

Source: `https://ieeexplore.ieee.org/document/6162911`

Fraga-Lamas et al. [4] evaluated the reliability of AR systems in the shipyard industry. The paper shows a collaborative AR system where an operator can place virtual notes upon production models and share them between coworkers. Another system allows users to visualize pipe plans and modify them in case of design errors.

The automotive industry was one of the first to integrate AR in its industrial process. As Abdul Halim reported in [6], early implementations of AR started in 2002 with the *ARVIKA* project that uses HMD and markers to recognize car components and superimpose on them the corresponding CAD model, plus other useful information. Just in 2006, a mobile markless AR system was proposed, it works with different

light conditions and is able to recover in case of tracking failures. *BMW*© realized a similar system in 2015, but it used a unique data goggle instead of mobile devices.

Segovia et al. [22] proposed an AR tool for production quality. Its usage, compared to a computer interface or handwritten data, provided the fastest and best results in term of interaction and problem solving of the users.

Figure 3.6: *BMW*© 2015 AR application

Source: `https://www.youtube.com/watch?v=P9KPJlA5yds`

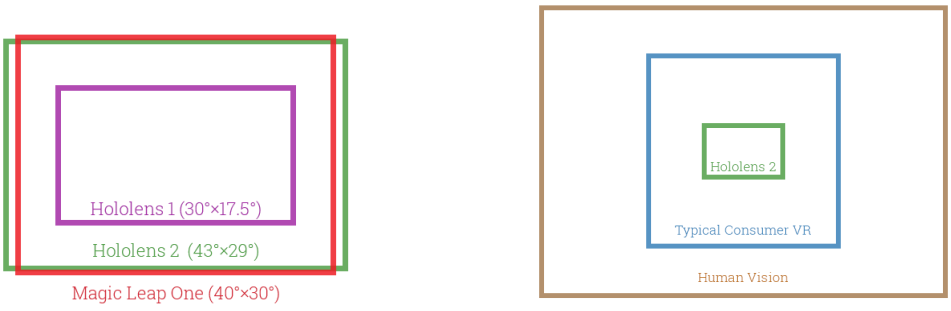
Figure 3.7: AR application for production quality

Source: [22]

A case study of AR for nuclear maintenance is described in [14], the aim was to help workers to see hazards that cannot been see by human eyes. This is crucial during the maintenance operation in which the radiation level can increase and is important to perform the task in the most efficient way possible.

3.2 Example of non AR monitoring system

Augmented Reality is a promising technology and will become an essential tool for industrial applications, but due to its low reliability and hardware capability it is not mature yet. Head Mounted Displays, such as *Hololens* or *Magic Leap*, have a limited FOV (field of view) compared to the Virtual Reality headset, and even more restricted if compared to the human one.
For this reason, aside from the cost of the devices, the usage of VR or desktop applications are also greatly diffuse in *Industry 4.0*. An example of a real-time monitoring system for a bridge structure was developed in 2010 and described in [20]. The acquisition of the 3D model of the bridge is performed using a point cloud data obtained by a three-dimension laser scanner, in this way is possible to collect data regarding changes on the surface and to convert them into displacement data.



(a) FOV comparison between AR HMD

(b) FOV comparison between AR and VR headset and human one

Figure 3.8: FOV comparison

Source: `https://casques-vr.com/`

Another one explained [37], shows a 3D Real Time system for cable temperature management. The system is browser-based and gives a 3D model of the monitored power well, for each wire is possible to know the position and value of temperature. The system also posses an alarm system based on the real-time data and send an alert with a pop-up message plus the position and identification of the cable.

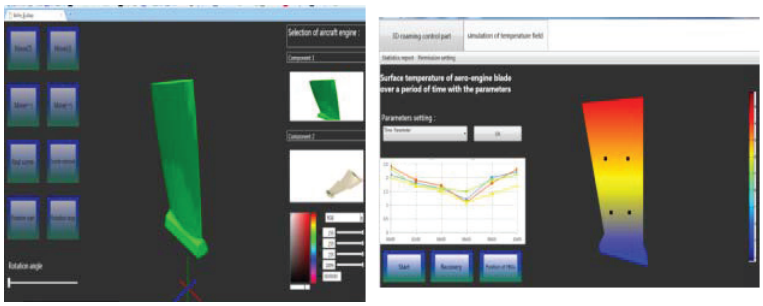


Figure 3.9: FBG bi-state monitoring system

Source: [11]

One of the aims of this thesis is to propose a 3D real-time monitoring system for FBG fiber, but it seems that no example of it already exists in previous articles. The only similar case is illustrated in [11], where a bi-state monitoring system was proposed for a blade in a rotating aero-engine. The system collects data from the FBG sensor and stores them into a database. Then a VR application, that works as a client, simulates the temperature of the blade based on the collected data, as well as the dynamic strain. The user can interact with the object by changing its rotation and the simulation it wants to perform.

Chapter 4

Proposed Real-Time Framework

The following chapter illustrates the general structure of the proposed framework describing the used technologies and the main motivation behind their selection.

4.1 General description

The real-time visualization framework, as illustrated in 1, is splitted into two applications.
The first one is a classic desktop program to be executed in any circumstance without the need for external devices. It was designed to avoid the user's need to modify the source code because it wants to change model, sensor position, texture, or network configuration. The second one was implemented to run on an AR environment; it is a "less-customizable" version compared to the desktop one due to motivations described in the following chapters. Despite its limitation, it permits the complete customization of the sensors upon the 3D model surface even if model and texture are static.

Both programs need to retrieve the most recent measures from the cloud platform, one for each active sensor. Because of this, they are designed to be multi-thread to maintain the GUI responsive. Communication between secondary threads and the main one, in charge of rendering and collection of user inputs, is permitted by a dispatcher system. It collects in a queue the operations that the main thread has to execute for the secondary one.

4.2 Unity3D engine

Unity3D is a cross-platform game engine, developed by *Unity Technologies*© and it was announced in 2005 during the *Worldwide Developers Conference*. At the moment, it is one of the most used engine for developing 2D, 3D, virtual and augmented games, simulations and other similar virtual experience. Even if it was born as a game platform, it is adopted as a production tool outside the game industry because it implements a wide number of features and extensions. Other than that, it

is supported in more than 25 platforms becoming the perfect tool if an application has to work on different devices.

Figure 4.1: Unity Logo

Source: `https://unity.com/`

A good definition of a game engine is *"[...] the software that provides game creators with the necessary set of features to build games quickly and efficiently"* [31]. Usually, the core elements of an engine are :

- **The Graphics rendering** dedicated to the visual outcome of an interactive application. Unity has its own real-time rendering pipeline that implements global illumination, ray-tracing, and physical based rendering. At the same time, it provides the tools to build a custom pipeline to be optimized for a particular platform or hardware.
- **The Physics system** used to handle physical simulations in real-time, in particular Unity uses the one developed by *NVIDIA©* called *PhysX©*, open source since 2018.
- **The Graphical User Interface (GUI)** and a set of tools (button, slider, etc) provided to build one.
- **The Scripting**, maybe the most important, used *" to define the logic of your game components by adding behaviours"* [31]. In this way, it is straightforward to control the elements in the scene and to create relationships between them. Unity offers a scripting API in C# that is converted to a C++ backend (IL2CPP) to increase the performance.

Nowadays, Unity is one of the best tools for developing AR/VR applications thanks to its cross-platform support; thus companies from more industrial fields such as automotive and aerospace had introduced it into their work pipeline. A manifestation of that is the *PiXYZ©* plugin that imports CAD models directly into the engine and handles different optimization operations that make the 3D model usable inside Unity for real-time developing [29, 31].

The fundamental element in Unity is `GameObject` that can represent anything from 3D objects, camera, lights, etc. It serves as containers of *components*, the functional parts of the engine, which describes how a `GameObject` should behave. By default, any `GameObject` has a `Transform` component that dictates the location, rotation, and scale of the `GameObject` itself. It can contain more than one components, each one with a set of properties that the user can adjust to achieve the expected result. Unity gives developers a set of built-in components that generate the main elements in a scene (the environment in which the application (game) is developed) such as cameras, lights, meshes, canvas, rigid object and more. Besides that, Unity permits to build custom components using the C# language. Exploiting the concept of inheritance of the OOP, the user is able to create a derivate class of built-in `MonoBehaviour` one that works as "blueprint". It can overwrite some particular functions that execute custom snippets of code in different moments of the Unity pipeline. The most commonly used are the `Update` one, called at each frame update, and the `Start` one, called before the application starts and is useful to variable initialization or similar operations [29].

Unity proposes three different plans depending on the user's need:

- **Personal**, a free licence, released for beginner or company with annual incomes less than 100$K.
- **Plus**, the less expensive licence, targeted to hobbyists or small companies with annual revenue of 100$K. Compared to the free options it gives learning course to master the engine and the *Unity Analytic* tool used to collect user feedback, crashes and exceptions in real-time.
- **Pro**, the more expensive licence, targeted to freelancers or professionals. It provided all the advantages of the **Plus** options plus custom support, priority access to Unity experts, and so more.

The project was developed using a **Personal** licence since its usage is limited only for research purposes. Besides, it is one of the most used tools for developing *Hololens* application [30], so forums and websites are full of guides and suggestions to help beginners in the learning phase.

4.3 Data visualization

One of the most important aspects of a visualization system is how data should be rendered to be simple to understand. Since the early development stages, the idea was to provide both textual and visual information to represent how the data are varying during the monitoring phase.

Textual data are the easiest to implements thanks to the GUI system offered by the *Unity Editor*. *"The Canvas component represents the abstract space in which the UI is laid out and rendered. All UI elements must be children of a* `GameObject` *that has a Canvas component attached"* [29]. The `Canvas` component specifies how the

GUI should be rendered with respect to the camera. The `Render Mode` property does this job for us, it can have three different values but only two are interesting for the purpose of this project:

- `Screen Space - Overlay`, the `Canvas` is scaled to fit the screen and the UI elements are placed on top of the scene. In this way, even if no camera exists, the UI is rendered and changes only in function of the screen's size/resolution.
- `World Space`, the `Canvas` is considered as a generic plane in the scene.

In the desktop application, the GUI is rendered with the `Render Mode` property set to the `Screen Space - Overlay` value . Thus, all the UI elements are treated as 2D objects making pretty simple the operation of translation on the screen space. With the support of the `EventSystem` component, any input devices data is collected and can be used to trigger different events. For the scope of this application, a simple message box, which contains the sensor's id and a wavelength value, was built. They are rendered upon all the scene's objects and near the screen position of each sensor. This grant the visibility of the text in any situation without the worry that other elements of the scene can hide it. To make the messages appear, the user can point the mouse's cursor on the sensor's area of interest or can click on a checkbox that displays all the message boxes at the same time.

On the other hand, the *Hololens* version's `Canvas` is rendered in `World Space`. Because of the limited FOV of the *Hololens*, it is necessary to not restrict it even more with the inclusion of a fixed GUI, in addition, the user could not always frame the monitored object. The message box levitates upon the position of the sensors and to make it visible is necessary to perform the air-tap gesture to the area of interest of the sensor. The message is always facing the camera direction; thus the displayed value is always visible even if the user is moving around the hologram. It could make all the messages visible at the same time by "tapping" on a checkbox.

4.3.1 Shader-base Heatmap

Even if textual data are essential, the user has to quickly understand how the data are evolving through time and what it is its relationship with the previous value. For this reason, visual information that involves colour applies perfectly on a visualization system.
One of the proposed features was to create a *heatmap* on the surface of the 3D model. To explain how it was developed, it is important first to know how meshes and materials work in Unity.

A 3D model, or mesh, contains a set of 3D coordinates (vertex) group together in triangles (a group of three vertexes). Each vertex carries additional information such as colour, normals (the direction that the vertex is facing) and 2D coordinate called UV. To be able to render any object in the scene, Unity required a Material component which wraps a shader that has a set of property that permits its customization [28].

Shaders can be described as a small program executed by the GPU in different points of the rendering pipeline. It is composed of small core units able to execute in a very efficient way calculation between vector, matrix, or a mix of the two. Thanks to the high level of parallelization that those type of hardware can offer, shaders are executed in an efficient and fast way and can manipulate an object's surface to behave in a certain way such as metal, diffuse or glass. A shader is executed for each pixel of the final render and is in charge of evaluating its final colour, which depends on the type of surface, light, and other properties.

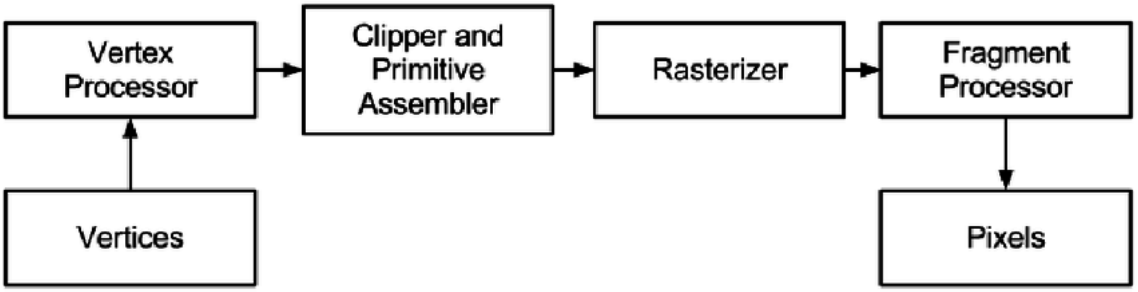


Figure 4.2: Example of rendering pipeline

Source: https://www.researchgate.net/publication/262398551_Cloud_and_Mobile_Web-Based_Graphics_and_Visualization/

There are different types of shades that take place in different points of the rendering pipeline. The most important ones are the vertex and fragment shader. The first one is called for each vertex, and its main purpose is to transform the coordinate from 3D world space to 2D screen one (the depth information of each pixel is inside a special buffer called Z-buffer). At this stage, colour, normals, position, and texture coordinate can be manipulated, for examples by light information. On the other hand, the fragment shader calculates the colour of each *fragment*, which is the data unit used to generate a single pixel. It receives as input the value calculated by the vertex shader and can perform some post processing effect.

Unity gives to the developer the possibility to write custom shaders with a similar C language that is successively converted into a native shader one such as GLSL, HLSL, Vulkan, Metal etc. The syntax contains mixed elements from the CG and the HLSL shader language. Unity's shader is divided into two main areas: *properties*, which are the input value of the shader, and *sub-shaders*, which contains the actual code. It is important to know that those *sub-shaders* are executed in a specific order, starting from the further object from the camera to the nearest one. The user can change the render order by exploiting the *Tag* properties inside any *sub-shader*. They can be used in different contexts to specify, for example, light information, transparency etc.

In the proposed visualization system, the heatmap was obtained with a custom vertex/fragment shader inspired by this tutorial [38]. The shader draws each pixel differently based on the vertex's distance from the sensor coordinates. The colours

are picked from a texture received as input. Each sensor is the centre of a sphere where the radius is the distance between the latest wavelength's value and the idle one.
When the *middleware client* sends a new packet, the application calculates a new radius and intensity, which defines the colour of the vertices that surround the sensor. Then, they are normalized to be inside a certain range and avoid that a single sensor can contain the whole mesh. At the end, they are saved as a global variable on the shader, declared inside it using the `uniform` keyword. Those are updated at execution time as `Material`'s component properties by a set of functions, which receive as parameters the property ID and the value to set. They accept base types such as `float` and `int` but also `structs` defined in a shader such as matrices, colours, textures and arrays. Setting and updating array are the central operations around this custom shader, all the heatmap's values (position and new radius and intensity value) are refreshed at the same time easily and compactly. The only flaw is that Unity uses the same HLSL principle in which an array's dimension is fixed after the first initialization.

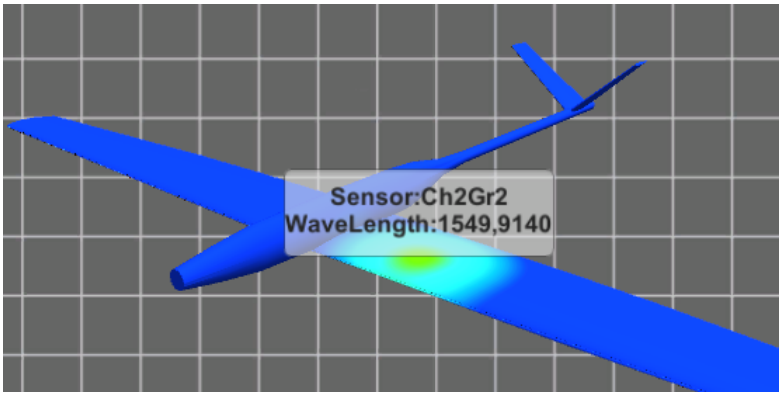


Figure 4.3: Example of the visualization system

4.4 Data collection

As the data visualization feature, the data collection one is a crucial element inside a real-time visualisation framework. The application should always be ready to collect new user inputs and to update the GUI, but at the same time, it must not freeze when it is waiting for new values from the server. The simplest way to accomplish this outcome is with a multi-thread program. The `.Net` framework offers different asynchronous programming patterns but the task-based asynchronous pattern (TAP), available since `.Net 4.X`, was chosen as the more suitable for a network communication and, more important, it is well integrated with the Unity engine.

Now the problem is to decide which layer of the previous framework should communicate with the visualization one. The following sections describe the two proposed solutions; it is given more emphasis on the one used at the end.

4.4.1 Mongo DB

MongoDB is a document-based database that stores data in a JSON-like format. It is classified as a NoSQL database which means that data aren't stored in the tabular format. As described in 2.1.4, it is part of *KaaIoT* platform and is rated as one of the best NoSQL databases for IoT system. In fact, it shows great results in term of response time, and throughput compared to other NoSQL databases when executing the "post" and "get" operations [3]. It is adopted as the main database for the cloud network layer because it was the only one able to manage the load from the middleware layer.

Microsoft© releases a *MongoDB* C# Driver as DLL. These can be easily added to a Unity project as *plug-ins* by just copying them into the project directory. The class `MongoDBManager` was developed to be in charge to manage the connection with the database. When the user confirms the network configuration, the `InitMongoDB` method is called to initialize the connection. In case of errors, no exception is thrown by the driver, so it was necessary to check if the database is reachable through a ping operation. This method is executed by the main thread to block any user input in case the configuration is not correct and avoid that it can start the simulation. The user can retrieve two kinds of information from the database: the sensors' configuration and the sensors' measurements. The first one is retrieved before the simulation start by calling the `RequestSensorsConfiguration` method. It check if at least one sensor is configured for the current monitored object and it is executed on the main thread for the same reason of `InitMongoDB`.
The simulation can start only if the database is reachable and the user decides launch it, only on this occasion a new thread is created. Its life span depends on the length of the simulation, and it is released only when it stops. The thread calls the `UpdateSensorInformtion` method, which retrieves the most recent JSON file and extracts all the measurements.

Because the C# driver is not well optimized, the application was not able to follow the huge amount of data produced by the interrogator and was generally very slow during execution. For this reason, it was abandoned after the first months of development. Even so, the class remained inside the final project even if the newest version was never tested.

4.4.2 TCP-Based Protocol

The *MongoDB* implementation alone did not reach the demanded specifications because of the poor optimization of the driver. Even if the database is able, on the middleware side, to manage the throughput of the interrogator, the driver interface was not designed to manage those amount of data at the requested speed. In our test case, the *SmartFibres*© *SmartScan* produces, if all 64 sensors are activated, 64 new peck data every 400 μs (called a *frame*). Each sample is encoded in 2 bytes and packed into a UDP datagram (up to 1500 bytes) where the first 36 bytes represent the header and the remaining one the payload. In the example configuration, a UDP packet contains up to 11 frames (each frame is 128 bytes). The maximum

transmission speed is 4.4ms because the interrogator wait to fill up the UDP packet before sending it.

Because of the high speed that the visualization system has to manage, the communication was conducted at a lower level of abstraction to achieve better control of the sent or received data. Beside, the middleware client and the real-time visualization framework didn't communicate directly. Thus, it could implicate additional delays between the two services. As a consequence of that, it was decided to make the visualization framework communicates directly to the middleware client through sockets. If the packet is forwarded to the visualization system without any delay, the data updates too fast to be readable by any human user. Because of this, only the last frame of each UDP packet was redirected. After the write operation on the socket, an additional 8 ms delay was added to ensure the readability of the samples. The final implementation managed to reach a speed comparable to the official *SmartSoft Application Software*.

It was decided to make the two layers communicate through TCP packets in a local network (LAN/WLAN), using a developed custom protocol. Because TCP is a stateful protocol, it guarantees that all packets reach destination without any concerns that they can be lost. This is crucial during the initialization phase, where both applications need to achieve a state in which they can send or receive data. When the simulation starts, a single socket is opened between the two, then, every time the middleware client receives a new UDP packet from the interrogator, it writes a new frame inside the socket's stream. In this way, the visualization system is notified only when new samples are available, and it has enough time to process them and to update the UI. The socket is closed only at the end of the simulation.

The custom TCP protocol defines five packet types:

- **DataPacket**: it contains the sample from the last frame.
- **RequestConfig**: it requests a configuration packet to the middleware client.
- **ConfigPacket**: it contains a list of all the active sensors for the current monitored object.
- **RequestDataStart**: it requests to start the monitoring phase.
- **RequestDataEnd**: it requests to stop the monitoring phase.
- **EndThreadPacket**: it is sent to stop the execution of the TCP server.

The content of the TCP datagram is divided into a header part, define by the id of the packet (32-bit integer value between 0 and 6) follow by the payload. For the `RequestConfig`, `RequestDataStart`, `RequestDataEnd` and `EndThreadPacket` type, there is no payload. Instead, the `DataPacket` and `ConfigPacket` have a fixed payload which contains the last frame and the active sensors' list, respectively.

The `DataPacket` payload is 768 bytes long and consists of a list of 64 elements which encoded two data, the sensor sample (encoded as a 32-bit float) and the scan

timestamp in μs (encoded as a 64-bit integer). In case the sensor is not active, both wavelength and timestamp are set to 0.

The `ConfigPacket` payload is 1600 bytes long. The aim of the `ConfigPacket` message is to let the visualization framework know which sensors are active before the simulation start. The payload is divided into 64 units of 25 bytes and each unit keeps the configuration for a single sensor. The 25 bytes are divided as described in the following table.

ConfigPacket Payload	
Bytes Range	Description
0-4	ID of the sensors which is a 32-bit integer inside a range of 0-16
5	Boolean value which indicates if the sensor is active or not
6-8	Sensor's wavelength in idle encoded as a 64-bit float. It could be zero even if the sensor is active
9-12	Sensor's wavelength maximum variation encoded as a 64-bit float. It could be zero even if the sensor is active
13-24	Sensor's position in world coordinate (x,y,z each one as 32-bit float) to locate it in the correct position on the model's surface. If the position is unknown, the sensor is located in the origin

Table 4.1: ConfigPacket payload description

4.5 Available AR technologies

One of the strongest points of the proposed real-time visualization system is the AR implementation. As the technology matures, new devices and SDK are announced to the public. In 2018, the *Magic Leap One* headset from *MagicLeap Inc.*© (announced in a demo in 2015) was finally available in the USA with a larger FOV, compared with other competitors. In the same year, the first official release of the Google SDK ARCore (and its equivalent iOS version ARKit) was finally released for a limited set of smartphones, and now, only one year later, the number of supported devices is duplicated. In late 2018, Microsoft itself announced the *Hololens 2*, it features a better hardware and a larger FOV compared to its first generation.

Among all the headsets available, the *Microsoft*© *Hololens* (first generation) was considered the most suitable device for this project because of its wide integration in similar industrial fields and the advantages of a headset compared to an mobile device. Other than that, it is well integrated to the Unity Engine thanks to the Mixed Reality Toolkit SDK. As described [32], the *Hololens* hologram's stability is quite good with a displacement error of 5.83 mm but if integrated with a mark-base system, like *Vuforia*, improve the accuracy on the horizontal plane when the vertical one has an accuracy of circa 4 cm [10].

Nowadays, the best solution to identify the position and rotation of an object in real-time is to use a marker-base system. Even if different deep-learning/machine learning solutions already exist for pose estimation and object recognition those aren't fast enough to guaranteed a stable frame-rate, especially for the *Hololens* hardware. Between all the market-based system available, the *Vuforia Engine* seemed the best solution thanks of its accuracy and good integration with both the *Hololens* headset and Unity Engine. Also, it is a tool extensively used and, thus, well tested.

In the following sections, there is an in-detail description of both the technologies.

4.5.1 Microsoft Hololens

Hololens is a pair of mixed reality smart glasses developed and manufactured by *Microsoft©*. Its tracking technology derives from the *Kinect*, a device also produced by *Microsoft©* for the *Xbox* gaming console, able to perform pose estimation using a set of RGB camera and an IR sensor. It was the first HMD compatible for the *Window Mixed Reality platform* under Windows 10.

Hololens features a set of high-definition lens, equipped with an optical projection system able to produce multidimensional images in full colour (holograms), able to mix the real world objects with virtual ones. To be able to understand the surrounding environment, *Hololens* is provided with an IMU (inertial measurement unit), which features an accelerometer, gyroscope, and magnetometer. In addition, it uses an *Intel Cherry Trail SoC*, which includes both CPU and GPU, and an HPU (Holographic Processing Unit), a co-processor able to perform spatial mapping and gesture recognition. The HPU and SoC have 1 GB of LPDDR3 each and share 8MB SRAM. Plus the SoC controls a 64GB eMMC and runs the Windows 10 operating system. *HoloLens* features IEEE 802.11ac Wi-Fi and Bluetooth 4.1 Low Energy wireless connectivity. It also is provided to an internal rechargeable battery, with an average life of 2 and half hours of active use [13, 36].

Exploiting the features of HPU, *Hololens* uses different types of inputs such as gaze, gesture, and voice, referred to as *GGV*. The **Gaze** tracks the position of the forward vector of the user's camera in the mapped area allowing the user to interact to the object it is pointing on. The gaze pointer is visible as a white circle that follows the head movement, similar to a mouse's cursor. Once the user targets an object with the gaze, it can interact with it using **hand gestures**. *Hololens* tracks either or both hands visible to the device if they are a ready stage (back of the hand facing the camera and index finger up) or in a pressed stage(same as ready stage but with index finger down). Those gestures are tracked inside a cone know as "gesture frame" that extends in all direction the display frame where holograms appear [13, 36]. The *Hololens* has two main gesture:

- *Tap-air* is a gesture which mimics a mouse click. It consists of a pressed stage (index up) followed by a release one (index down). With it, it is possible to execute a hold, manipulate and navigation operation on the hologram object.

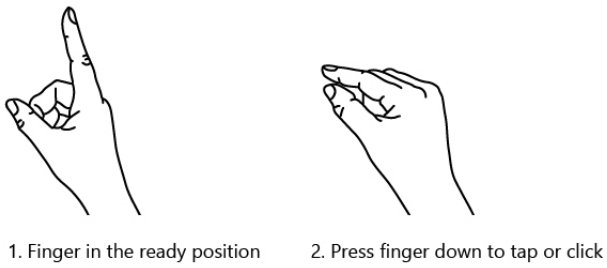


Figure 4.4: Steps to execute the Air-Tap gesture

Source: [13]

- *Bloom* is a special gesture that is used to open the start menu of Windows 10 when an application is running. *"To do the bloom gesture, hold out your hand, palm up, with your fingertips together. Then open your hand"* [12].

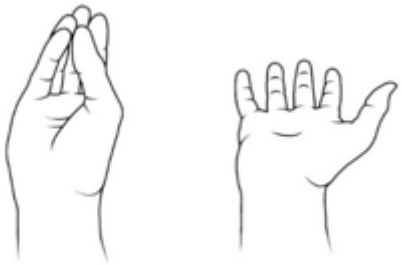


Figure 4.5: Steps to execute the Bloom gesture

Source: [13]

4.5.2 Vuforia

Vuforia Engine is the most used platform for AR development because of its wide support in different devices such as smartphone, tablet, eyewear, and platforms like Android, iOS, UWP and Unity Engine. The API is released in C# for the Unity Engine and in C++ for the other ones.

It supports different type of object recognition that can be summarized in two main categories: **Images** and **Objects**.
Image targets is a market-based tracking system. The engine tracks the main features of an image that are stored inside a database. The database can be download and used, as an additional package, for runtime comparisons. It supports PNG or JPEG both in colour or grayscale with a maximum size of 2MB.

Model Target is a markless tracking system that uses the object's geometry to locate it in space. The trackable object should be rigid, fixed in the environment its

(a) Image Target Example

(b) Model Target Example

Figure 4.6: Vuforia Engine examples

Source: `https://library.vuforia.com`

locate into and with recognizable surface features. A model target is created using the Model Target Generator (MTG) application, which is provided as part of the *Vuforia Engine SDK*. It takes as input a 3D model of the tracked object, and then it checks its suitability. Then, the user chooses a *Guide View*, a frame of the 3D model where it can change angle and distance from the camera. After that, it generates a database that can be used as a package inside the user's application. Additionally, it can train the database using a deep-learning cloud service where the model is trained to recognize different *Guide Views* or all object's sides.

Vuforia Engine also provides an extended tracking functionality which maintains the model into the scene even if its target is no longer in the field of view of the camera. It is integrated with devices such as *Hololens* or smartphone which support ARCore/ARKit SDK because they can perform spatial mapping or plane detection [8].

Chapter 5

Implementation Details

The following chapter describes the implementation details of the proposed framework. It starts with the common features on both applications and then it details the elements only available on the desktop and the *Hololens* implementation.

5.1 Common features

During the initial design of the two applications, it was clear that they should both follow a general structure and code implementation because the two development platforms are quite different from each other. In particular, they differ in input devices (or methods), user interaction with the virtual environment and hardware capability that required to design different strategies to obtain the best result in term of usability and performance to follow the requirement dictated by the middleware client. After in-depth considerations of all the available possibilities, it was decided to design a set of elements in a way that they required the minimum set of discrepancies between the two implementations and, in general, didn't need modifications at all. These features are explained in the following sections.

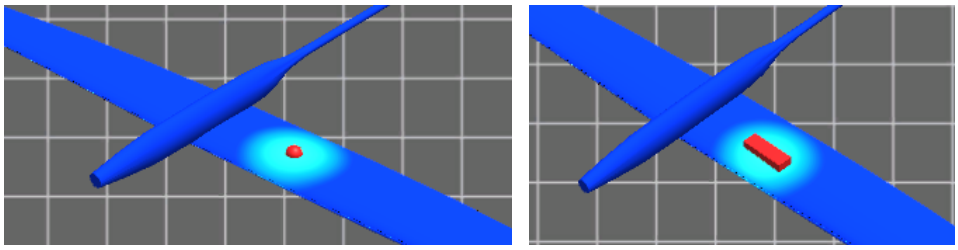
The primary choice was to specify different states in which an application should be. These can change in number between the applications, but two are used in the whole framework: `Menu` and `Monitoring`.

```
1 public enum Status {
2     MONITORING,
3     MENU
4 }
```

Listing 5.1: Status enum description

During the `Menu` status, the user can interact with the GUI and set different configurations regarding the network, the insertion, modification, and elimination of sensors inside the system and the type of measure it wants to perform. For the last option, the user can choose between only two types of measure: temperature and displacement. The first one represents a sensor with a sphere and the second with a parallelepiped. Furthermore, the user can also modify the position of the

sensors upon the surface of the 3D model. In case the user wants to measure the displacement of the object, the rotation of the sensor can also be adjusted. It rotates around the normal of the plane in which is placed.



(a) Temperature sensor model (b) Displacement sensor model

Figure 5.1: Representation of the sensors in the framework

Instead, in the `Monitoring` state, the framework receives data from the middleware client (or retrieved from MongoDB). The user cannot modify any configurations and it is only able to visualize the data on the model as textual or visual information.

5.1.1 Class structure

A significant portion of the code was shared between the two applications splitted in different classes. These define the parts of the structure used to make the primary and secondary threads to communicate (`UnityMainThreadDispatcher`), to control the monitored object status and the configuration of its sensors (`MonitoredObject`), to support the socket interaction with the middleware client (or *MongoDB*) (`TCPManager` and `MongoDBManager`) and to manage the GUI (`GUIManager`). At last, one class (`GameManager`) is in charge to direct the other ones and to handle the application's status and error messages between the different threads. All of this classes extend the `MonoBehaviour` Unity class, so they are represented as objects that have a location inside the scene. Let's go in order and explain each class in details.

The `UnityMainThreadDispatcher` class is in charge to contain a list of methods requested by the secondary thread to be executed by the main one (source code [18]). It implements a `singleton` pattern, which means that only one instance of that class exists for all the running time of the application. The secondary threads save each request in an `Action` object and call the methods `Enqueue`. It inserts the object inside a *Queue* collection, which guaranteed the first come, first serve rule. Inside the `Update` function, the less recent `Action` is dequeued securely and invoked on the main thread. This is reassured by the fact that the `Update` method, of all the classes that extend `MonoBehaviour`, is called inside the *Unity Pipeline*, which is executed in single-thread.

The `MongoDBManager` class is in charge to manage the communication with a *MongoDB* database inside the cloud network platform. Its implementation is explained in 4.4.1.

```
public class MongoDBManager : MonoBehaviour {
    ...

    /// Method that initialize the mongo's variable
    public bool InitMongoDB () {...}

    /// Method to get the sensors configurations
    public void RequestSensorsConfiguration () {...}

    // Method to start thread that get data sensors from database
    public bool StartRequestSensorData () {
        ...
        Task.Run(() => UpdateSensorInformtion());
        ...
    }

    /// Method that retrievex samples to the database
    void UpdateSensorInformtion () {...}

}
```

Listing 5.2: `MongoDBManager` class

The `TCPManger`, similar to `MongoDBManger`, regulates the communication with the middleware client through TCP packets. The connection is via LAN/WLAN, so both of the system should be connected to the same network. To support this, the `Awake` function, executed before the application starts, saves all the unicast addresses associated to a set of the network interfaces available on the system inside a list. This set contains only interfaces that can transmit data packet and it excludes all the loopback and virtual interfaces. When the user inserts and confirms IP and port of the middleware client, the method `InitTcpListener` is executed. First, it checks if the entered address has the same subnet of one of network interface and, if at least one exists, it performs a ping operation to see if its reachable. In case everything goes right, it creates two TCP listeners on two separate thread: the first one listens on the port value entered by the user (wait for configuration packet), and the other one listens on the previous port value incremented by one (wait for data packet). In the case of a negative outcome, the application shows an error message and invites the user to insert the configuration again.
One thread executes the `ListenerConfig` method and the other one the `ListenerData` method. Both remain active until the `TerminateThread` bool variable is set to false. When the user wants to change network configuration or to close the application, the variable is set to true and guaranteed that the threads are released without the worry of resource leakage. A loopback packet is sent to wake a listener in case is waiting for a TCP client.
`ListenerConfig`, as the name suggests, wait only for `ConfigPacket`, it calls the `ServeConfigPacket` to parse the content of the custom TCP payload and then it advises the main thread to save the update configuration.
Instead, `ListernData` waits only for `DataPacket`. When a new client establishes

a connection, it calls `ServeClientData`. The method cycles until the client's network stream contains at least one frame. For each message, it examines the header first and, if is correct, it parses the content, otherwise discards it. When parsing the message, it updates not only the information concerning the active sensors but also calculate data for the visualization system. The radius and intensity of each sensor is determined based on the maximum variation from all sensors, which guarantee a coherent heatmap visualization. At the same time, it updates a sensor's configuration in some particular circumstance:

- The sensor's idle value is zero. The physical system is on a resting stage before starting a monitoring phase, and so the value measured by an FBG sensor. Thus, the first received data becomes the idle value.
- The sensor's maximum value is smaller than the one measured. The application updates the sensor's variation and the global variation to normalize the heatmap.

At the end of each message, `ListernData` advises the main thread to update the GUI and the monitored object's heatmap.
When the user wants to request a new configuration or to start the monitoring phase, it sends a TCP packet to the middleware client. The `Send` method performs the previous operation; the type of message is specified by the input argument `type`. If it is not correct, the application displays an error message.

```
1  public class TCPManager : MonoBehaviour {
2      ...
3      List<string> allLocalIP = new List<string>();
4
5      public void Awake () {
6          // Get LAN address
7          ...
8      }
9
10     /// It start the TCPListener based on the information setted by
       the user
11     public bool InitTcpListener() {
12         ...
13         Task.Run(() => ListenerConfig(AddressInfo.Port));
14         Task.Run(() => ListenerData (AddressInfo.Port+1));
15         ...
16     }
17
18     /// TCP Server for the application, it run on a different
       thread
19     public void ListenerData (Int32 port) {
20         ...
21             while (!GameManager.instance.TermianteThread) {
22                 using (client = server.AcceptTcpClient()) {
23                     ServeClientData(client);
24                 }
25             }
26         ...
```

```
27      }
28
29      /// TCP Server for the network configuration
30      public void ListenerConfig (Int32 port) {
31          ...
32          while (!GameManager.instance.TermianteThread) {
33              using (client = server.AcceptTcpClient()) {
34                  ServeConfigPacket(stream);
35            }
36          }
37          ...
38      }
39
40
41      /// Send message to the interrogator client
42      public void Send ( TypeMessage type) {...}
43
44      /// Serve data packet
45      void ServeClientData ( TcpClient result ) {...}
46
47      /// Serve configuration packet
48      void ServeConfigPacket ( NetworkStream stream ) {..}
49
50  }
```

Listing 5.3: `TCPManager` class

Since now, the named classes don't change between the two application because their implementation works in both development platforms. Instead, the following ones are implemented quite differently except for few methods.

The protagonist of the framework is the `MonitoredObject` class, which manages the visualization of both textual and visual information on the 3D model and the user interaction depending on the application's status. Also, it is in charge to update the sensors upon the model's surface in terms of position, rotation, size, and models. Inside the `Awake` function, it prepares the 3D model by merging the different parts in a single mesh and it extracts information about the sensors' position by looking for parts with name `SensorGR_CH` where *GR* stay for the grating and *CH* for the channel of the sensor.
Each time the sensor's configuration is updated, the `SetSensor` method is called to update the sensor's model position, if it already exists, or to create a new one. The value saved inside the configuration dictates the position of the sensor, but, if it is not specified, the application choose a random point on the object's surface; in both cases, the method doesn't update the rotation because there is no field for this value inside the sensor's configuration packet. The user can also change the sensor's model (`ChangeTypeSensor`) and its size (`ChangeSizeSensor`).
The class generates, for each sensor, a message box used to visualize the data during the `Monitoring` phase. When a user tries to interact with the sensor, or with its influence's radius, the `ShowText` method pops up the panel near the pointed position. If a user decides to see all the message boxes at the same time, the application

calls `ShowAll` instead. In the `menu` status, the `HideAll` and `HideText` methods are invoked to hide all the notes.
The main aim of `MonitoredObject` is to update the shader's heatmap. Each time a new frame arrived at the `ListenerData` thread, the `UnityMainThreadDispatcher` calls `UpdateShader` which receives in input two lists: one of the new properties (radius, intensity) and the other with the last frame's samples. The class maintains a reference to the `Material` component, and so it immediately updates the shader with the function `material.SetVectorArray("_Properties", newProperties)`. Besides, if the sensors measure displacement, the method gives the sensor a different color if it is loose or pull. Another essential operation that the class performs is to update on the sensors' position inside the heatmap in case the monitored object is rotated or translated.

```
1  public class MonitoredObject : MonoBehaviour {
2
3      public Material material;
4      ...
5      void Awake () {
6          // Get all the object's meshes, combine them and extract's
       sensors' position
7          ...
8      }
9
10     void Start () {...}
11
12     void Update () {
13         // Update position of the sensors if the main object was
       moved
14         ....
15     }
16
17     /// Update Heatmap properties
18     public void UpdateShader ( Vector4[] newProperties, List<
       KeyValuePair<UInt64, float>> wav ) {...}
19     }
20
21     /// Update sensor information
22     public void SetSensor ( Vector2 defaultProperties ) {...}
23
24     /// Change size of each sensor
25     public void ChangeSizeSensor (float newSize) {...}
26
27     /// Change type sensor based on the type of measurament
28     public void ChangeTypeSensor () {...}
29
30     /// Show panel with sensor information
31     private void ShowText ( Vector3 point, string nameCollider )
       {...}
32
33     /// Show all sensor's panel
34     /// </summary>
35     private void ShowAll () {...}
36
```

```
37      /// Show panel with sensor information
38      private void ShowText ( Vector3 point, string nameCollider,
        List<KeyValuePair<UInt64, float>> wav, Vector4[] newProperties )
        {...}
39
40      /// Show all sensor's panel
41      private void ShowAll ( List<KeyValuePair<UInt64, float>> wav)
        {...}
42
43      /// Hide all the sensor's panel
44      private void HideAll () {...}
45
46
47      /// Hide panel when outside of object
48      private void HideText () {...}
49
50  }
```

Listing 5.4: `MonitoredObject` class

The class `GUIManager` is used to control the main graphical interface, activate/deactivate parts of it, based on the user's choices and the application's status, and verify the input values. The base schema of the UI used for this framework is the following one.

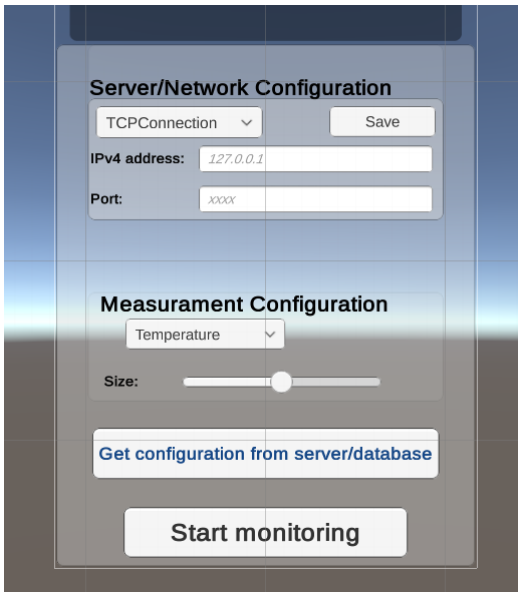


Figure 5.2: Main framework's GUI elements

It includes three main areas: *Server/Network configuration*, *Measurement configuration* and two buttons used to contact the server/database.
The first lets the user chooses on a dropdown menu between `TCPConnection` and `MongoDBServer`; the first is the default option and provides two input field where IP and Port can be entered. In case the IP has an incorrect format, an error message

is displayed. Depending on the selected option, the `GUIManager` rearranges the panel's size and shows additional input fields (Database and Collection name). A *Save* button is located on the same high as the dropdown menu. If no data is entered, the user cannot interact with the other parts of the GUI because the class's `Start` function disables them. Only when it inserts a functional configuration and it clicks on the *Save* button, the `ConfirmNetworkConfiguration` method enables the other parts of the GUI, disables the network one except for the confirmation button, substituted with the *Cancel* one. The *Measurement configuration* part shows a dropdown menu with the temperature and displacement option, which switch the sensors' model, and a slider for changing the size. At last, there is a *Get configuration from server/database* button to request a `ConfigPacket` and the *Start Monitoring* which starts the `Monitoring` phase. To stop it, the user has to interact in different ways based on the application's platform.

```
public class GUIManager : MonoBehaviour {
    ...
    void Start () {...
        // Set default value on input field
        SetDefaultValueInputField();

        // Disable the sensor and measure menu
        ...
     }

    /// Actuvate/Deactivate sensor's configuration button
    public void ToggleSensorConfigurationMenu(bool value) {
    }

    /// Set the current configruation on GameManger
    public void ConfirmNetworkConfiguration(GameObject button)
    {...}

    /// Change GUI depending on the type of network configuration

    public void ToggleNetworkConfigurationMenu(bool value) {...}

    /// Set default value on the network configuration input field
    public void SetDefaultValueInputField() {...}


    /// Change rectangle size of the Network UI + activate/
    deactivate option correalted only to the database one
    public void ChangeNetworkMenu() {...}

    /// Change IP address: in case of an invalid one an error
    message is
    public void ChangeIPAddress() {...}

    ...
    /// Other input check methods
    ...

```

```
}
```

Listing 5.5: `GUIManager` class

Last but not least, the "director" class `GameManager` manages the "life" of the entire applications and works as intermediate for inter-class connection. Like the dispatcher, it implements a *singleton* pattern and it is the first custom script to be executed in the *Unity pipeline* to ensure a correct initialization of all the application's variables. It contains a set of methods that can be divided into three main categories: network configuration, sensor configuration, components and status managing. Also, it acts as a "local" database for a set of global variables affordable by any element of the framework at any moment. It guarantees the end of any secondary thread by updating the `TerminateThread` flag.

```
public class GameManager : MonoBehaviour {
    ...
    /// List all classes instances
    ...
    public bool SetConfiguration { get; private set; }
    public List<Sensor> SensorsFromNetwork = new List<Sensor>();
    public Status statusGame { get; private set; }
    public volatile bool TermianteThread = false;
    ...
    public static GameManager instance = null;

    /// Set singleton and init values
    Awake{...}
    ...
    // Network Configuration Methods
    //-------------------------------------------------
    NETCONF_METHODS
    ...
    // Sensor Configuration Methods
    //-------------------------------------------------
    SENSORCONF_METHODS
    ...
    // Component/Status managing Methods
    //-------------------------------------------------
    STATUSMANAGING_METHODS
    ...
}
```

Listing 5.6: `GameManager` class

The network configurations methods manage the interaction between the `GUIManager` and the `TCP/MongoDBManager` instances. Depending on the user actions and input value, it sets the `SetConfiguration` property and applies or cancels the previously confirmed setting with the methods `StartUp` and `ShutDown`. Both of them are called by the `ConfirmConfigurationNetwork` method, used for saving or deleting a network configuration. When the user changes IP, port or others parameters, the

equivalent `Update`*`Parameter`* method substitutes the old setting with the new one and forwards the configuration to the `TCP/MongoDBManager` instance.
The sensor configuration methods, instead, are the middle chain between the `TCPManager`/ `MongoDBManager` /`GUIManager` classes and the `MonitoredObject` one. Each time the user adjusts the senor's position or request a new configuration, it is called `UpdateSensorInfo` in charge to create or update a list, which collects the sensors' configuration. If necessary, it notifies the `MonitoredObject` to update the model's positions. During the `Monitoring` phase, the dispatcher is used to call `GameManager` methods to forward the data directly to the `MonitoredObject` which update the heatmap and panels with ad hoc-methods. At last, it manages each component's life cycle directly using `Init`*`ClassName`* methods. The passage between the `Menu` and `Monitoring` status (and viceversa) is executed by the `Start/EndSimulation` functions. The global flag `statusGame` dictates the current's application status and forces the classes to behave differently after a new phase is set.

To conclude, the framework represents a sensor's configuration as a `Sensor` struct. It contains id, channel, idle wavelength, maximum wavelength variation, position plus some additional data calculated to speed up the control operations.

```
public struct Sensor {
    public float MaxWavelenght;
    public int SensorID { get; set; }
    public int Channel  { get; set; }
    public bool Active  { get; set; }
    public float WavelenghtIdle { get{...} set{...} }
    public float MaxVariation { get{...} set{...} }
    public Vector3 Position { get{...} set{...} }
    ...
}
```

Listing 5.7: `Sensor` struct

5.1.2 Multi-thread communication

It was mentioned previously that the framework works on a multi-thread environment. The main thread is in charge to execute the *Unity Pipeline* and, so, it manages GUI and user inputs where the secondary ones are in charge of the network communication. The dispatcher list and the shared variables of `GameManager` are the only resources shared between the multiple threads, which it means they need to be protected to guarantee mutual access and avoid data corruption. Depending on which network configuration the user chooses, the framework works in two different multi-thread system.

If the user wants to connect to a *MongoDB* database, there is only one new thread during the `Monitoring` state. After a new JSON file is parsed, the network thread calls the dispatcher to send the last frame to the `GameManager` in charge to update the shared sensor's list. During this operation, the concurrent list access is secure thanks to lock statements used in the get and set operations.

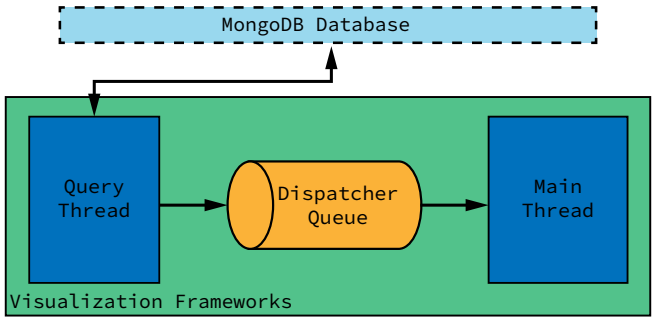


Figure 5.3: Multi-thread connection schema for *MongoDB*

If the user wants to adopt the TCP connection, the application manages two different thread that works as servers. These are instantiated when the user confirms the network settings and listen in two different ports. One handles only the `ConfigPacket` when the other one waits only for `DataPacket` messages. They work in two separate game status, so they don't interfere with each other. In this way is easier to manage the application's concurrency and at the same time increase the parallelism. Similar to the *MongoDB* situation, the shared data are protected by lock statement.

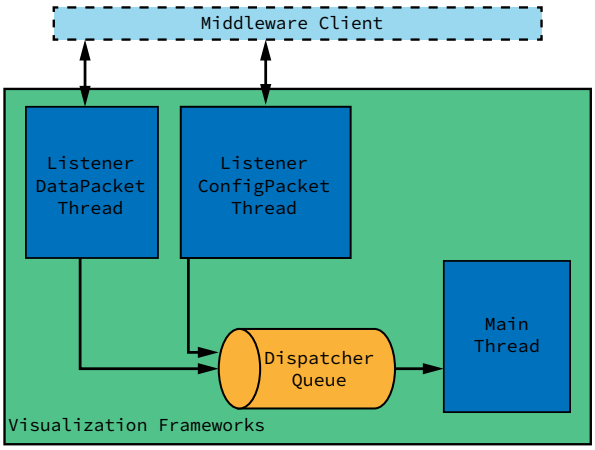


Figure 5.4: Multi-thread connection schema for TCP Connection

5.1.3 Heatmap Shader

As previously described, the heatmap shader receives in input a texture, the number of active sensors, their positions in word space, and their properties split into radius and intensity. The shader calculates the distance between the current vertex and the sensor coordinate, it normalizes the value based on its radius and it gets the color intensity from the texture, according to its intensity.

Unity takes advantage of the HLSL build-in vectors created form basic types. A 3D vector, for example, is saved as a `float3` with x,y,z components when the 2D equivalent (`float2`) only has two (x,y). Those structures are usually used to save colors and coordinates, but a programmer can exploit them to store additional parameters that the shader needs to perform its task. In this case, the sensor's properties utilize a 2D vector type to save radius and intensity.
As previously described, a vertex and fragment programs compose the heatmap shader.
The vertex one receives as input the `appdata_base` struct defined in `UnityCG.cginc` which holds raw data correlated to position, normal and one texture coordinate of the current vertex. The program calculates the world and screen coordinate, and the color for each vertex. In particular, the color is estimated from the vertex normal and some built-in light variables declared in `UnityLightingCommon.cginc` file. All of these parameters are saved inside a `vertOutput` struct which is the output value of the vertex shader. The value after ":" are HLSL semantics that represents individual mesh's data elements.

```
struct vertOutput {
  float4 pos      : POSITION;
  fixed4 diff     : COLOR0;
  fixed3 worldPos : TEXCOORD1;
  fixed2 uv       : TEXCOORD0;
};
```

Listing 5.8: vertOutput struct description

`vertOutput` is the input argument of the fragment shader, which is the one actually in charge to generate the heatmap. The program cycles to all the active sensors and calculates the influence of each one based on how far it is from the current vertex. If it is inside the influence's radius, it calculates the intensity and accumulates it to the `h` variable. The value summed in `h` corresponds to the i-th width pixel of the input texture.

```
half4 frag(vertOutput output) : SV_TARGET {
  // Loops over all the points
  half h = 0;
  for (int i = 0; i < _Points_Length; i++) {
    // Calculates the contribution of each point
    half di = distance(output.worldPos, _Points[i].xyz);
    half ri = _Properties[i].x;
        ...
    h += hi * _Properties[i].y;
  }
  ...
  half4 color = tex2D(_HeatTex, fixed2(h, 0.5f));
  color *= output.diff;
  return color;
}
```

Listing 5.9: Fragment shader original code

In the original implementation, there was no smooth transition between two values, so, to obtain a better visual result, it was developed an alternative version. The shader receives two properties arrays, one including the previous frame and the other the last frame. It lerps (linear interpolate) between the two based by an interpolant value update directly by `MonitoredObject` on the `Update` function.

```
half4 frag(vertOutput output) : SV_TARGET {
  // Loops over all the points
  half h = 0;
  for (int i = 0; i < _Points_Length; i++) {
    ...
    half ri = lerp(_OldProp[i].x,_NewProp[i].x, _Blend);
        ...
    h += hi * lerp(_OldProp[i].y,_NewProp[i].y, _Blend);;
  }
  ...
}
```

Listing 5.10: Fragment shader modified code

The visual result was a soft animation between the two heatmap, which was satisfying to watch. Unfortunately, this version doesn't work properly for fast data rate since the application doesn't have the time to refresh the interpolant properly.

5.1.4 Middleware connection

To include the update visualization framework into the IoT one was necessary to adapt the middleware client to communicate with it correctly. For this reason, a new version of the client was developed, which maintains the same structure of the one implemented for the [5].
It was realized in `C++` for `Linux` with the aim of maintains the same performance in terms of speed and throughput. To achieve this goal, two additional threads were introduced to manage the conversation with the new visualization framework: one runs the TCP server to listen for data and configuration packet requests or data end transmission requests, the other sends frames as a continuous network stream for the duration of the monitoring phase. On the original middleware, the interrogator interface already talks with the device through UDP datagrams, so it already exists all the necessary mechanism to guarantee a correct concurrent execution. However, to facilitate the implementation and because the application didn't require a strict real-time implementation, the `C++11` thread core feature was used instead of the `phthread POSIX` API since it gives a higher level of abstraction to the developer without the risk of losing performance. The class defines a set of `atomic boolean` values as global variables to save the user's request and guaranteed the correct termination of the threads avoiding memory leakage. To manage the mutual access to shared resources a `C++ mutex` was used. The usage of `lock_guard` and `unique_lock` wrapper classes ensure that the lock is released a the end of a code block delimited by curly brackets.

For each frame of the packet, the main thread saves the samples inside a local

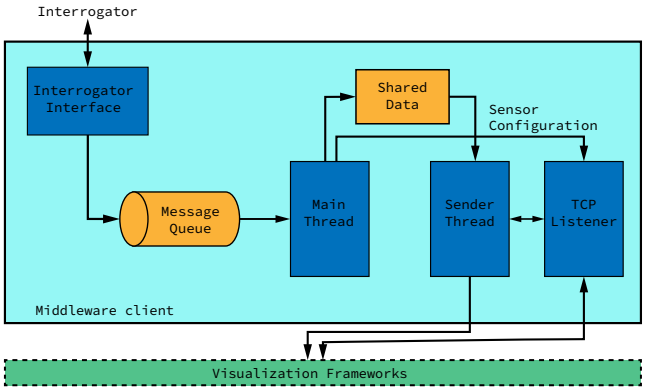


Figure 5.5: Multi-thread connection schema on middleware client

vector, but only the last one is copied inside a global vector shared with the "sender" thread. This one instantiates a new socket to the TCP server of the visualization frameworks only when the middleware client receives a `RequestDataStart` packet. Thus, the server sets the `atomic boolean send_data_holo` to notify the "sender" thread to create it and to write a new `DataPacket` message each 8 ms. When it receives an `RequestDataEnd` packet, the thread closes the socket and waits for a new request. Instead, a `RequestConfig` is managed directly by the server's thread. The first frame is utilized to extract the current system configuration by identifying the activate sensors and save them inside a list of `sensorConfig` struct. The application terminates when it receives a `SIGINT` (interrupt signal) and force all the tread to terminate in a controlled and clean way.

5.2 Desktop solution

The framework was first implemented as a desktop application but designed to reuse the code for the *Hololens*'s implementation. It was developed and tested on a *Windows* environment but thanks to the Unity cross-platform feature there is no problem to be ported for other OSs. It implements a broad set of new procedures and extends the common ones. A desktop interface is more known for a user to interact, so it was easier to implement new features in a way that the user feels it natural and intuitive. Also, desktop OSs has a very established way of accessing the file system and installing new programs, even if now users are more familiar with mobile interfaces than the one offered by a classic personal computer. At the same time, it is the most comfortable platform for implementing a new application because it permits to both debug, test and develop the program on a "native" environment.

5.2.1 Overview

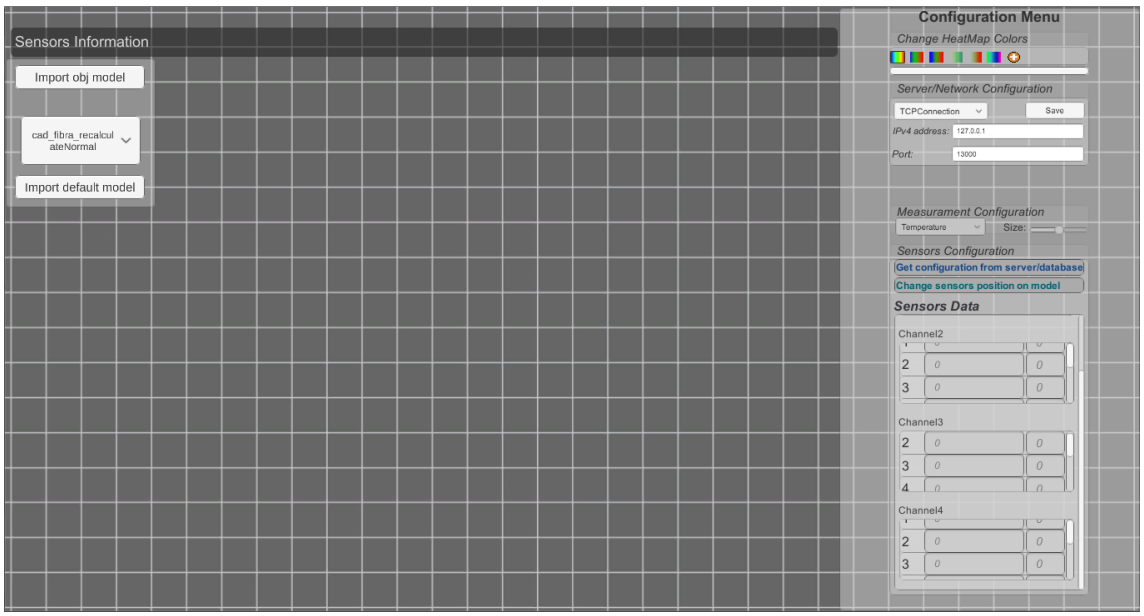


Figure 5.6: Desktop application home view

The previous image is the file home screen of the desktop application after it started. It is divided into three main areas.
The left section, called *Configuration Menu* extends the standard GUI (5.2). From the top, it contains a new section called *Change HeatMap Color* composed by a list of different color gradients with a `plus` symbol at the end. At the base, it shows the *Sensor Configuration* panel which includes the already mentioned *Get configuration from server/database* button. Aside from that, an additional button is introduced (*Change sensors position on model*) to make the user update the sensors transformation and a scrollable view, called *Sensor Data*, arranged in four main sections, one for each interrogator's channel. Each segment is also a scrollable view composed of sixteen rows. A single row holds a number (the id of the sensor inside the channel), the idle and the max variation value (both input fields). The right part is entirely new and includes two buttons (*Import obj model* and *Import default model*) and a dropdown menu with a list of different preloaded 3D models. The central part, instead, is left free because it is the portion of the screen that the 3D model should occupy. Upon it, there is a gray bar that is used to display error and debug messages useful not only to a developer but also to a user as textual feedback.

By just locking to the desktop interface is evident that the users have more options from the base implementation. During the design phase, it was clear that some structural elements need to be refreshed to assured a better role subdivision and code organization inside the program.

Therefore, two new statuses were introduced. First, the `Menu` status was split in two: `Menu` and `ChangePos`. Now the `Menu` one works as an "idle" mode where the user interactions are limited only to the GUI and can't no more approach the 3D

model in any way. Instead, `ChangePos` is activated after the push of the *Import default model* button. Here the user must not interact with any UI elements but only with the 3D models, for this reason all menus are hidden. It can interact with the sensors' models and changing their position and rotation as it wishes. Besides, the user can also rotate the object around its pivot by simply clicking on the mesh's surface with the mouse's left button and hold it when moving it in any direction; as result, the object rotates toward the cursor's last pointed coordinate. In this way, it is easier to find the correct spot in which locate the sensor and its presented in a very intuitive and straightforward way so anyone doesn't have any problem to remember how to do it just after the first try.
The new status, `Import`, is pretty straightforward. In fact it allows the user to import either a `.obj` model by selecting it from the file system or one of the four models already part of the application, called "default". The right side of the GUI permits the switching back and forward to this status.

5.2.2 In game features

By the previous section, it is clear that each new graphical element corresponds to new features:

- The user can select the 3D model of the object directly inside the application.
- The user can modify the idle and variation value of the sensors.
- The user can customize the heatmap colors.

A detailed description of each point is located in the following subsections

3D model import

As described in the previous section, the new status `Import` lets the user select any `.obj` model from the file system. In this way, there is no need to recompile the program each time it is required to change the monitored physical system. This is an advantage because the end user doesn't need to learn Unity or to coding to apply this small modification, plus errors and bugs caused by the lack of experience are entirely avoided. To realize this feature was necessary to use two external tools to reduce the developing time: one recreates a GUI interface for the file system during the execution time [25] and another automatically import a `.obj` as a `GameObject` [24]. Those are managed directly by the `GUIManager` using the menu on the right. When the user clicks on the *Import obj model* button, a file browser interface appears. It shows only `.obj` file and directories, but it is simple to extend it to any formats by adding the new extension to a list.
Here, the user can decide to select a file or go back to the main screen, but it cannot interact with any other GUI elements to avoid errors or failures. After the file selection, the `LoadFileUsingPath` is called to perform the actual import by the `ObjLoader` class which parses the file and extracts all the essential data such as vertex, normals, triangles and more to represent a 3D model into the scene.

```
1     ...
2     public string[] fileExtentions = new string[] {"obj"};
3     ...
4
5     /// Call the File Browser prefab and instantiate it
6     public void CallerFileBrowser () {
7         ...
8         FileBrowser fileBrowserScript = fileBrowserObject.
    GetComponent<FileBrowser>();
9         fileBrowserScript.SetupFileBrowser(ViewMode.Landscape);
10        fileBrowserScript.OpenFilePanel(fileExtentions);
11        ...
12    }
13
14    /// Load a file if the path is not null
15    private void LoadFileUsingPath ( string path ) {
16        if (path.Length != 0)
17            GameObject obj = new OBJLoader().Load(path);
18        ...
19    }
20
21    /// In case the browser is closed the black panel is deactivate
22    private void CloseBrowser () {...}
23
24    /// Reset import menu Gui
25    public void ResetImportMenu() {...}
26
27    /// Load prefabs
28    public void LoadDefaultPrefab (TMP_Dropdown value) {...}
29    ...
```

Listing 5.11: `GUIManager` import methods

After that, the `MonitoreObject` class is in charge of executing different procedures on the mesh, fufilled by `OpenNewMonitoredObject`. To also avoid possible errors caused by interaction with other GUI elements, beside the import one, the *Configuration Menu* (left part) was hidden. The application should ensure that the object is always big enough to be visible on the center of the screen and composed by a single mesh to easy the manipulation during the `Monitoring` and `ChangePos` phase. First, the `MonitoredObject` transform component is reset and the rendering one disable. The current sensor's configuration and models are withdrawn and the new object's reference is saved into a global variable. Then, it seeks for mesh components inside the children of the imported object (if they exist) and combines them into a single mesh (`CombineMeshes`). The resulting one usually doesn't have the pivot in the correct position, to know how far it is from the actual center of the mesh the application exploits its bounding volume, simplified as a box. From it, Unity calculates its center; if it is different from a zero vector, any single mesh's vertex is translated which permit the change of the pivot's coordinate (`CenterMesh`). Because the size of the 3D model could be huge, with respect to Unity's virtual world, it's necessary to normalize the volume to be inside a cube of 1.5 unit long. It is achieved by the

method `NormalizeMesh`. Firs it extracts the higher dimension of the bounding volume and divides every single vertex coordinates to that value. The original center and the max dimension are saved into global variables because they can be reused to transform the sensor's position arrived from the middleware client since they are correlated to the original pivot and size. However, `ManageImportMonitoredObject` doesn't have only to "normalize and center" the model, but also extracts possible information regarding sensors position and ID. It was one of the most delicate features to implements caused by the huge number of events that can occur during the previous operations. Each sensor is identified by a specific label which associates the 3D coordinate to the grating and channel of the sensor (`Sensor`*`GR_CH`*). After unifying the sensor's object into a single mesh and extract its center, it necessary to calculate the global position of the sensor. Because during the mesh combing the resulting mesh could be mirrored along one of the three axes (or a mix of them), a `XOR` bit-wise operation is executed between the coordinate of original vertex and the one from the combined mesh (before all the transformation) to check if they have the same sign or not. The final sensor's coordinate is calculated in the following way.

```
1     ...
2     sensor.transform.position = 1.5f * ((sensorCenter -
    originalCenter) / maxDimension);
3     ...
```

Listing 5.12: Calculation of sensor's position

At last, the original sensor's objects are destroyed and substituted by a compliant model provided by the visualization system.

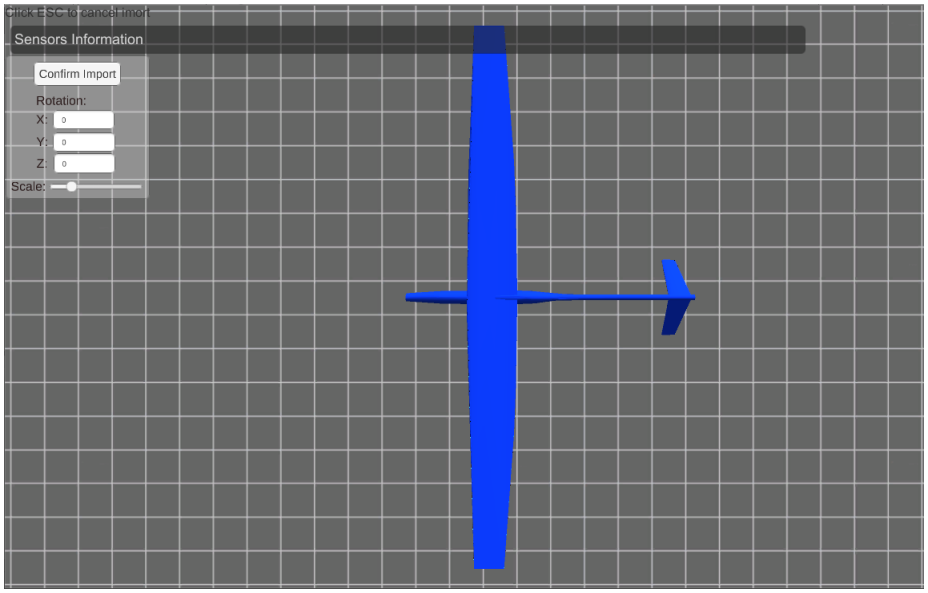


Figure 5.7: Parameters to customize during the `import` status

Afterward, the user sees a new GUI where it can specify some parameters before it can confirm and end the model's import. It can control the mesh's rotation and

size. Three different input fields identify the rotation parameters (x,y,z), and a slider distinguish the size one. These avoid strange effects when turning the model around its pivot. If it diverges from the center of the object (corresponding to the center of the screen) and the object doesn't face the camera, the interaction with the 3D model results confusing and unpleasant for the user because of the mismatch between action and feedback on the screen.

The user can undo the import procedure any time by simply clicking the `ESC` key. Instead, it pushes the *Confirm Import* button to confirm the parameters. Then, the `GameManager` is notified and concludes the import phase. Before returning on the `Menu` status, the object needs to be transformed once again based on the user's inserted data. As before, the operation is executed per vertex. If applied on the `Transform` component, the effect could not be pleasant because, in its "idle" status, the object isn't facing the right direction. For confirming the sensor's position and rotation, a new model is instantiated for each sensor. The final mesh is assigned to the `MonitoredObject` object and the imported one destroyed.

```
1     ...
2     /// Start import new Monitored object model (only .obj format)
3     public void OpenNewMonitoredObject ( GameObject sourceObject )
    {...}
4
5     /// Update rotation of the object.
6     public void UpdateRotation ( TMP_InputField field ) {...}
7
8     /// Update local scale of the object
9     public void UpdateScale ( Slider slider ) {...}
10
11    /// Method that combine the mesehes
12    private void CombineMeshes ( List<CombineInstance> list,
    GameObject objectC ) {...}
13
14    /// Normalize the mesh of the GameObject
15    /// </summary>
16    private float? NormalizeMesh ( GameObject objToNorm ) {...}
17
18    /// Resize the mesh
19    private void ReduceSizeMesh ( GameObject objToNorm, float size
    ) {...}
20
21    /// Center the mesh
22    private Vector3? CenterMesh ( GameObject obToCenter ) {...}
23
24    /// Confirm the model imported
25    public void ConfermModel () {...}
26
27    /// Stop the import procedure
28    public void CancelImport () {...}
29    ...
```

Listing 5.13: `MonitoredObject` import methods

Heatmap customization

In a visualization tool it is fundamental to customize the color of the heatmap or graph. If the user is more familiar with a set of color is correct that it can use it; also it should be able to switch between different palette depending on the information that it wants to see. The desktop application supplies this possibility. In the top part of the left panel, it was added a menu called *Change HeatMap Color*. If clicked on the different gradient square, the user can change the heatmap color instantaneously to other default palettes; the selected one is identifiable by a black outline. However, if it clicked on the *plus* symbol, a new interface appear:

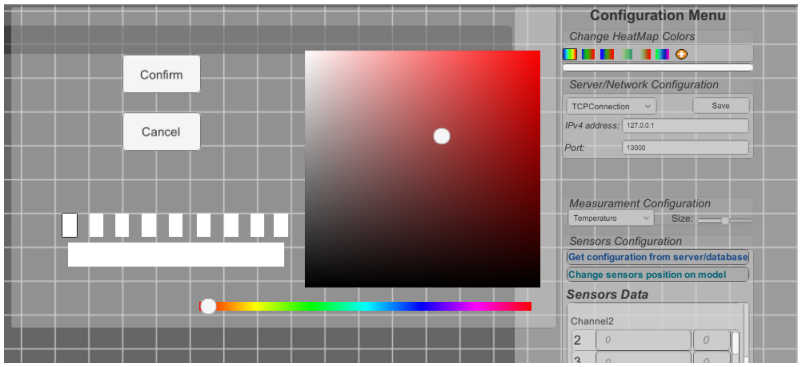


Figure 5.8: Custom HeatMap colors GUI

When this new panel appears, all other GUI elements are colored in gray to represent the fact they are disabled. The current big square represents the selected hue with all the possible combination with white and black (value). The hue can be changed by the slider below. On the right, there are two buttons: one confirms the chosen heatmap colors, the other cancels the operation. At last, there is a white rectangle with upon nine smaller one which divided it in section. When clicked upon the big square, a new color appears on the outlined box. The bigger rectangle is refreshed on the corresponding position with the same color blended between the two white sides. If clicked on another small rectangle, named *gradient color*, it is possible to execute the same operation and, so, create a new palette. After decided which colors are adapt or preferred for the new palette, the user has only to click on *Confirm*. Then, a new gradient appears at the top of the default squares, outlined in black and applied on the 3D model.

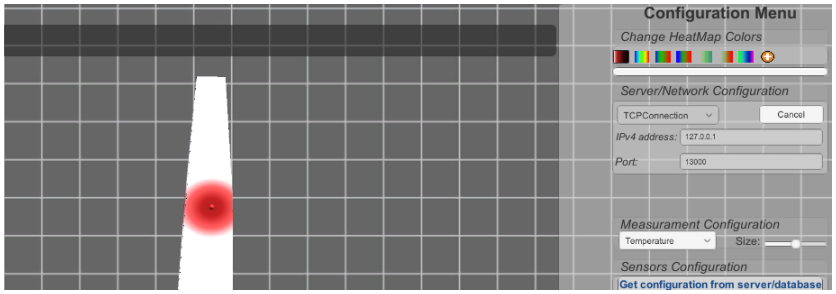


Figure 5.9: Update heatmap on the model and UI

The heatmap customization is available only in `Menu` mode. Two classes, independent by the main application structure, control its behavior. `TextureManager` manages the GUI menu, it collects the event of every single *gradient color* and updates the view every time a new colormap is available. It is also responsable to show the Color-pickers GUI and disable the other GUI elements. On the `Awake` function it uploads the default heatmap as `Image`'s component inside a *viewport*'s content. They are automatically imported as `Texture2D` objects, so there is no problem to use them as a parameter for the shader.
`GradientGUI`, instead, handles the Color-pickers GUI and the back-end logic. Inside the `Start` function, it creates for each hue value a `Texture2D` with all the white and black shades. In this way when the user changes hue the texture is already calculated and just needs to be swapped, without the worry of any additional delay. It manages a `Gradient` object divided into nine main sectors (associated with an index). When the user selects a *gradient color*, the application defines which section of the rectangle needs to be updated. To pick a new color, the function `PickColor` extracts the color information of the pointed pixel inside the 2D texture and use it to update the gradient.

Sensor position and orientation customization

One of the most significant features mentioned was the possibility to customize the position and rotation of sensors upon the 3D model's surface. The desktop implementation had a new status (`ChangePos`) dedicated to conducting the task exploiting the `MonitoredObject` class to implement it.

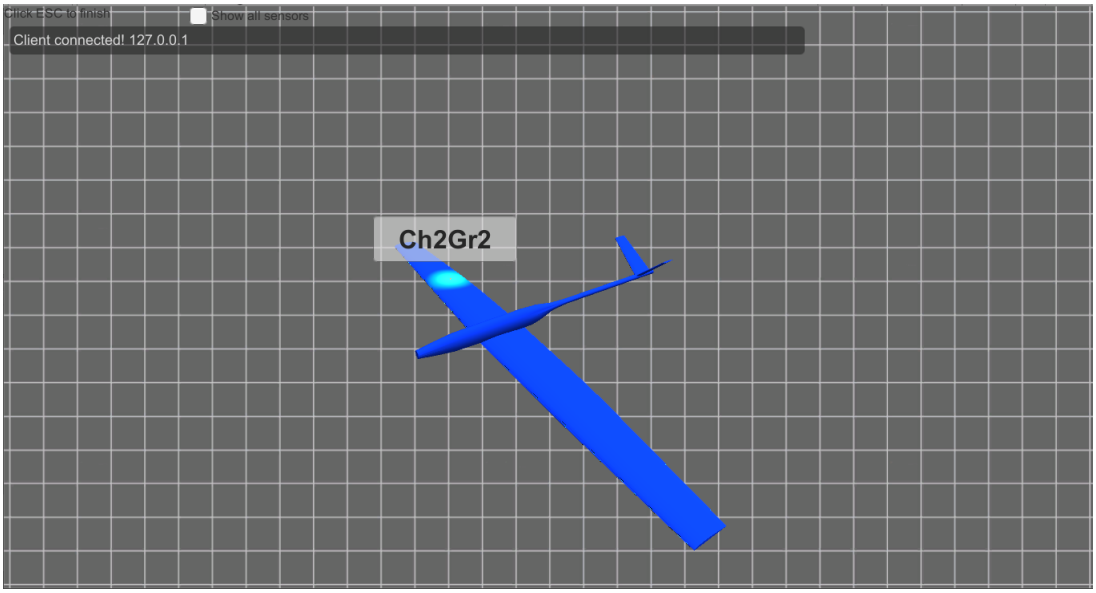


Figure 5.10: `ChangePos` status main screen

The pressing of the *Change sensors position on model* button starts the `ChangePos` mode, but if no sensor is active on the surface, the application doesn't change status and notifies the user with an error message. Otherwise, the main GUI is hidden and

the only form of interaction is with the mouse on the surface of the model. The only textual information is the message upon the error panel which alerts the user to use the `ESC` key to return in `Menu` mode and a check-box used to show all the sensor's ID. In fact, each sensor is provided with a panel to display its ID composed by this label `Ch`*`X`*`Gr`*`YY`*; for example, a sensor located in the first channel with grating two has an ID equal to `Ch1Gr2`. The user can view the message box by hover the mouse's cursor upon the sensor's model or clicking on the mentioned check-box. In case the user rotates the monitored object, the panels are developed to follow the sensors' position at each `MonitoredObject`'s `Transform` adjustments.
If a sensor's position needs to be updated, the user clicks with the left mouse button on the sensor's 3D model, if picked a glowing red aura surrounds it. Instantly the model follows the mouse cursor upon the system's surface and with just another left-click it is released. During all the "picked" period the sensor has the glowing effect to alert the user which one it has selected.

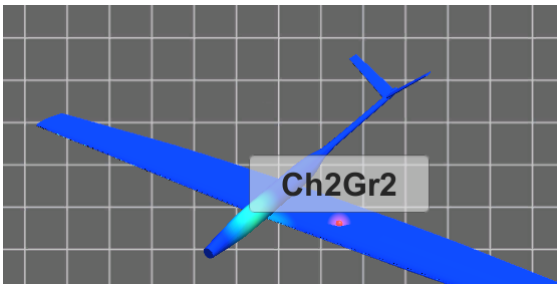


Figure 5.11: Selected sensor's glowing effect

The implementation exploits the usage of the `Raycast`, which cast a ray from the camera's position in direction of a pointed coordinate. It stops when it hit a `GameObject` equipped with a `Collider` component used to check physical collisions, which can also comprehend situations where the user wants to interact with a particular object. The `MonitoredObject` is provided with `MeshCollider` which matches the shape of the monitored object's mesh perfectly. Even the sensor's model has its own collider used to be recognized by the ray during the selection phase. When selected, the sensor's collider is disabled so it can follow the the object's surface. If omitted, it climbs back the ray until it reaches the camera. The model's correct position is essential, but it feels rough since it doesn't follow the surface's curvature and sometimes intersects it. The effect is quite visible if the sensor's model is the parallelepiped, where half of the was not visible. Therefore, the model should be rotated accordingly to the surface's curvature. It was used the normal of the hit point and the up vector of the world space to reach the desired result.

```
1    ...
2    Transform t = selectedSensor.transform;
3    t.position = hit.point;
4    t.rotation = Quaternion.FromToRotation(Vector3.up, hit.normal);
5    ...
```

Listing 5.14: Selected sensor's model position update

Another import detail is the sensors' orientation on the surface. For displacement's measure, it is fundamental to know the pointing direction of the fiber to have a visual clue about its path and, as a consequence, the orientation of the displacement. The user can accomplish this operation by holding down the mouse's right button on the sensor's model. In this case, the object should rotate around the normal of the surface in which is placed. Each vertex of a mesh is associated with normal vector (vertex normal) calculated from other vertexes contained in the same triangle. Within the `RaycastHit` parameters, there is also the index of the poked triangle of the collider's mesh. From there, it is straightforward to extract the vertex index, and so the normal's value. These parameters are only available if a `MeshCollider` is used; therefore, after the first hit, the sensor's model collider is disabled. When the user releases the right button, the collider is immediately re-enabled.

```
1    ...
2    Transform t = selectedSensor.transform;
3    MeshCollider meshf = GetComponent<MeshCollider>();
4    Vector3[] triangles = meshf.sharedMesh.triangles();
5    Vector3[] normals = meshf.sharedMesh.normals();
6
7    Vector3 normal = normals[triangles[hit.triangleIndex*3]];
8    t.RotateAround(pivot, transform.rotation * normal, 1);
9    ...
```

Listing 5.15: Selected sensor's model rotation update

Sensor data customization

The previous feature permits the customization of position and rotation of the sensor upon the system's surface, but there no option to delete or add new ones. Without these possibilities to change the sensor's number, the modification of their transform's properties is pointless. The middleware clients indeed provide all the necessary information about active sensors but its not uncommon that some sensors could be broken or not present inside that configuration.
For this reason, it was designed a set of custom GUI elements to represent all the possible sensors supported by the *SmartScan* interrogator. In total it contains 64 sensors divided into four different sections (maximum the number of channels), each one containing 16 sensors (also called grating, the maximum number of sensors supported on a fibre). The *Sensor Data* part in the left side of the main screen represents the described elements. Each section's sensor is created at run-time by the `GUIManager` and by default represents an inactive sensor. Each sensor is a "row" of three elements: the ID (grating) and two input fields for the idle wavelength and maximum wavelength variation, the last estimated by the user or obtained after some tests. If the user fills at least one of this two, the application set that sensor's as active and generates a new model upon the object's surface. To eliminate it, the text in both fields should be canceled. Then, the sensor is considered as inactive and the program deactivates the corresponding `GameObject` to be reused in case of a user's error.

This part was developed during the early stages of the program's life and it became less relevant going further in time because of some late update it both the middleware client and the visualization framework. It was preserved not only for the sake of integrity but also because it was the fastest way to add sensors to debug or test the application.

5.2.3 Real time graph

One of the late features of the framework was the introduction of a real-time updated graph. The end users need to study the behavior of the sensor in different circumstance and displaying only the last measured wavelength wasn't very useful in this context. A sample without any back history is just a number but if correlated to past or feature measurses it starts to acquire value. One right way to see how a sensor evolves during time is to use a line graph, that represents in the x-axis the time and y-axis the wavelength value. The graph feature wasn't developed from scratch, but it was integrated using an external plug-in called *Graph and Chart* [23]. The reasons are simple, fist it is the best tools in Unity for this purpose and also supports real-time insertion for a high data-rate. Last, the time to implement even a simplistic version of it wasn't enough. So, instead of including an in-development functionality, it was adopted something already tested by a vast number of developers. More information about the tool can be found in its documentation [9].

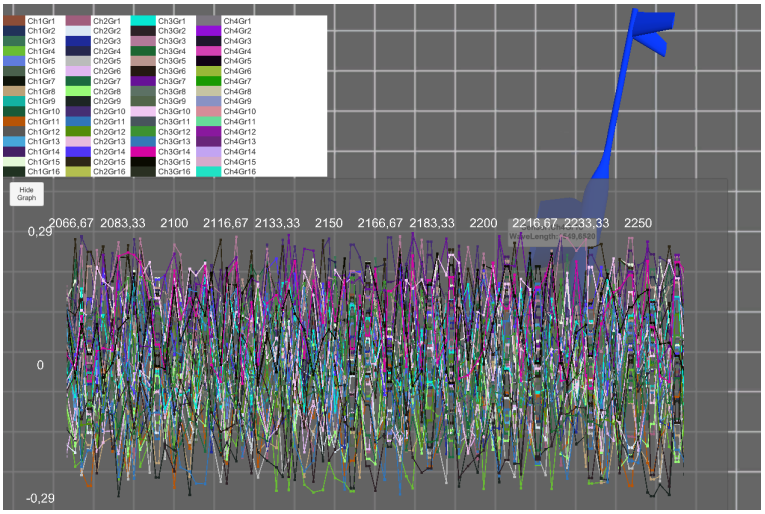


Figure 5.12: Example of the real-time graph with 64 sensors

For this project, it was chosen to use a line graph. The plug-in permits its customization by script, without the need to always use the *Unity Editor*. With a handful of functions it possible to add new categories, new points and customize the horizontal or vertical view to automatic resize the graph or to scroll the horizontal view after it reaches a given number of points per category. The plug-in also creates a legend for each line represented inside the graph. The `GUIManager` class contains different methods that permit the initialization and the update of the graph based on the number of active sensors. The `InsertCategory` method adds new categories

linked with active sensors, in case the user switch between two sensor's configuration it removes the ones that are no longer active. Each category is distinguishable by a different color that is picked randomly with the method `Random.ColorHSV()` offered by the Unity Engine. The method saves the number of active sensors and then decides to make the views scrollable after a given number of samples. If the number is lower than 40, it accepts until 20.000 points per line, but if it is bigger than 40, only 200 points per line are visible on the screen. Even if it seems unreasonable, this guaranteed that the application could maintain an acceptable frame-rate and performance in terms of speed and memory usage. Besides, it avoids that the application crashes if a line reaches the maximum number of vertex permitted for a mesh inside Unity. When a new `DataPacket` is parsed, the `UpdateCategory` method is executed with the update wavelength samples as input. Each sensor's frame is associated with a timestamp that represents the instant in which the interrogator scans the fibres and retrieves the peak wavelengths. Because it is measured in microseconds, the application uses it as a unique ID and discards possible duplicates received from the interrogator. It updates all the categories with wavelength value different from zero and shows only the variation between the new sample and the idle one. The x-axis doesn't show the timestamps, but the time-span between the first received data and the newest one converted in seconds.

```
1     ...
2     /// Insert a new category on the Line graph
3     public void InsertCategory() {...}
4
5     /// Update the line graph, a point for each active sensor
6     public void UpdateCategory ( List<KeyValuePair<UInt64, float>>
      wav ) {...}
7
8     /// Set the line graph as visible
9     public void ShowGraph() {...}
10
11    /// Set the line graph as not visible
12    public void HideGraph() {...}
13    ...
```

Listing 5.16: `GUIManager` graph methods

It is given the user the possibility to hide the graph by simply clicking on the *Hide Graph* button below the graph's legend. The same operation on the *Show Graph* one restores the default's view.

5.2.4 End simulation features

The real-time visualization system provides a simple data analysis tool at the end of each monitoring operations that received at least one packet from the middleware client. The tool provides two main features: a log file and a line graph. Both are managed by an additional class called `LogManager`. The `GameManager` is in charge of calling its initialization method `IntLog` before the `Monitorng` phase starts. It creates the log file and creates an additional thread to write the data asynchronously

to the rest of the application. All the received samples are saved into a shared queue called `LogQueue` by the `TCPManager` (o the *MongoDB* one). Instead, the `WriteOnLog` method retrieves and writes them in order into the file. The log is written in a CSV format and with the following template:

```
Timestamp       ,Ch1Gr1    ,Ch2Gr1    ,Ch3Gr1    ,Ch3Gr2
1560958460391038,1547.6090,1545.9410,1548.1930,1539.00
1560958463950843,1547.6090,1545.9400,1548.1940,1539.00
1560958467510150,1547.6080,1545.9400,1548.1940,1539.00
1560958471069954,1547.6100,1545.9400,1548.1930,1539.00
1560958474629745,1547.6100,1545.9390,1548.1940,1539.00
1560958478189062,1547.6080,1545.9400,1548.1950,1539.00
```

Figure 5.13: Example of a Log file with five active sensors

The operation continues for all the `Monitoring` phase and only when the user switches to the `Menu` one the method exits from its while loop. It extracts all the remaining measures from the queue to complete the log. After that, it closes the file and executes an external program on a different thread using the `Process` class functionalities. The program consists of a short `python` script that includes the `Numpy` and `Pandas` library, widely used in the machine learning and data science field for data analysis. In this case, it was used for its straightforward methods to retrieve all the measures from the CSV file and build a line graph, successively saved as a PNG image. The operation is not immediate but requires a few seconds. After that, the image is available on the build directory of the framework. At the end of the script's execution, both threads terminate.

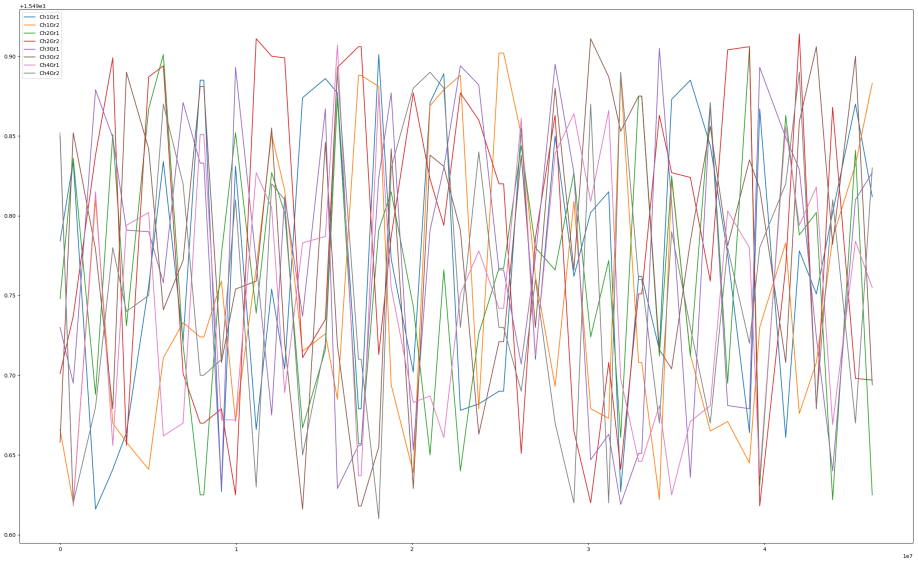


Figure 5.14: Example of line graph generated by the python's script

Because in Windows doesn't have a default `python` interpreter, it was used the *Anaconda* distribution that automatically sets the `Windows` registers to execute the `phyton.exe` command anywhere on the command line.

5.3 Hololens solution

As described in the chapter 3, AR technology is widespread in the various industrial field for its advantage of superimposing virtual information on a real environment. In maintenance and monitoring task, headsets show better user's performance and focus. For these reasons, it was decided to implement the application for the *Hololens* platform instead of a port to other hand handled device ones. Compared to a general desktop interface, *Hololens* offers a different approach, especially in terms of input capability. Even if a "mouse clicker" device exists, the user interacts more likely using hand gestures or voice input, because it is more intuitive and doesn't require nothing else than the headset itself. However, it creates a more satisfying immersive experience, thanks to the spatial mapping and the accurate hologram anchor system. The user can move far for the hologram and continue to see it in the same position even after sudden head movement.
The *Hololens* port uses the majority of the standard architecture of the visualization platform, but some procedure required small adjustments. The different interaction system and the possibility to navigate through the "mixed reality" world required to think the interaction system over to guaranteed the same level of comfort when using the interface or carrying out tasks.

5.3.1 MRTK v2 Release Candidate 1

During the first months of design, *Microsoft*© announced the new *Hololens* model (called 2) that could be released in the late 2019 or 2020. As a consequence of that, a new SDK was developed by the *Microsft Mixed Reality Toolkit* (MRTK) team to create a unified set of features and components for Unity to help developers to build application supported by a various set of devices such *Hololens* (model 1 and 2), Windows Mixed Reality headsets and OpenVR headsets (HTC Vive / Oculus Rift) [26]. It is the successor of the original *Hololens Toolkit* that didn't receive any update since 2017.

Figure 5.15: MRTK logo

During the development of the application, MRTK V2.0 betas and refreshes were released, but it was decided to develop the application using the RC1 release (Release Candidate 1) which is described as an intermediate version between the beta and

the official release. On the GitHub page, it is reported that is pretty near the release stage if not for small bugs and small finish touches. On the current state, July 2019, the RC2 was released.

The usage of the new MRTK 2.0 was essential to support a complete (or partial depending on the SDK bugs) porting to the new *Hololens* device which should solve the significant interaction problem encounter in this implementation. In general, the new SDK gives all the useful tool to implement the port in a simple way. One of the most significant advantages was the inclusion of an emulator that can be activated directly on the *Unity Game Player*. With it, is possible to run the application on a similar environment before tests it directly on the *Hololens* headset. Another useful component adapts the Unity UI elements into *Hololens* thanks to the `Canvas Utility` one plus additional scripts that easy the interaction with different objects on the mixed reality when near them. With a set of interfaces, the user can implement how an object reacts depending on the type of interaction like pointers, gestures, voice, etc. In the *Hololens* port, the user can interact with GUI and 3D model only with the air-tap gesture, considered a pointer's type.

5.3.2 Limitation

Nowadays, the *Hololens*' hardware is a little bit outdated and it wasn't sure if it was capable of handling all the desktop features. Then, the limited FOV permits that only a small area supports the holograms, reducing the situation in which the application could be used. In fact, the application isn't able to frame entirely large objects or building but just a small portion of it, which makes difficult to understand what it is inspecting. To avoid low frame rates and, as consequences, low usability of the device, only the sensor's number customization was fully implemented. The model's import was discarded because of the limited capacity of the *Hololens*'s flash memory and the huge number of space that a 3D model can occupy. The heatmap customization was considered at first but then discarded because it doesn't consider essential for the port.

Even if the MRTK Unity plug-in is the best available tool for developers, it has some limitations. Since the first *Hololens* SDK the user doesn't have the raw access of the camera stream and the sensor's data directly in Unity but should uses external tools or third-party library. So it wasn't an easy task to perform computer vision operations such as object and image tracking.

5.3.3 Overview

The *Hololens* port, as previously announced, goes through a series of small changes due to platform issues.
First of all, the main GUI is untouched if compared to the default one but follows the user inside the virtual world. This was decided to not limit the user's FOV even more due to the already limited visual capability of the device. It is possible to reduce the dimension of the GUI to a little square by just clicking on the `minus` button. All the elements are hidden except for a black panel used to display errors

and messages and another `plus` button used to restore the `Canvas` at its default state.

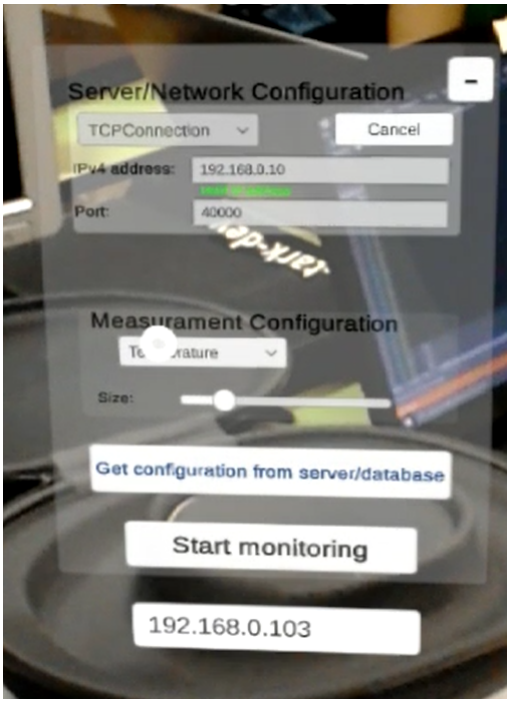


Figure 5.16: *Hololens* main GUI

The `Canvas` always follows the user around the scene by positioning on the upper right of the screen when the camera stops moving. On the Unity Editor, the `Canvas` rest location can be changed by using a drop-down menu on the `GuiFollowCamera` component. It provides other customizable parameters such as movement speed, maximum and minimum distance from the user. This implementation isn't enough to guaranteed good user experience. In fact, the *Hololens*'s *gaze* works as a mouse cursors and it is always at the center of the screen. So, every time the camera moves the `Canvas` does the same, therefore clicking any UI elements becomes a chore. To avoid this unpleasant pattern, the `GuiFollowCamera` extends not only the `MonoBehaviour` but also the `IMixedRealityFocusHandler` interface. It defines how an object reacts to a focus enter/exit, or in other words, how an object reacts when the gaze points on it. In this case, the usability is implemented in a straightforward way, until the user is facing the `Canvas`, it stays still. However, when the *gaze* focuses on another object, it starts to follow the camera again. In this case, the interaction is more manageable and intuitive, especially for someone who never used an AR headset before.
Another problem to solve was the insertion of the network configuration. By default, the input field works with a physical keyboard and don't automatically create a virtual one. To solve this issue at every `InputField` object is provided with two main components: `InputFieldHolo` and `Event Trigger`. The second is used to call the `OpenSystemKeyboard` method when the user performs an air-tap gesture on a `InputField` object. It opens the system's virtual keyboard and disables the

`Canvas`'s `GuiFollowCamera` component to avoid any difficulty caused by the camera's movement. When the user clicks outside the keyboard or cancels the operation, the system hides the keyboard but doesn't disable the object. This can cause failures when the user wants to open the virtual keyboard in a second moment. As countermeasure both the `CloseKeyboard` and `Update` methods set the keyboard status as inactive before reactivating the `GuiFollowCamera`'s component.

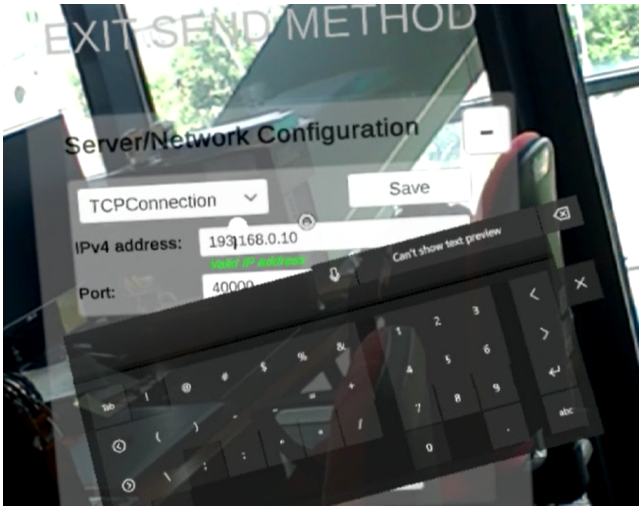


Figure 5.17: Hololens's virtual keyboard

All the *Hololens* application are built as UWP, means that the application is converted into IL2CPP back-end when it needs to be uploaded on the device. Unity includes a *Windows Runtime* support that let the user calls native system WR API to execute some code snippets. In the current application, it is used to retrieve the network interface's local IP because `System.Net.NetworkInformation` is not fully supported in UWP. The desktop application uses the `NetworkInterface.GetAllNetworkInterfaces()` method to return a list with all the available network interfaces but in WR API the `NetworkInformation.GetHostNames()` is adopted instead.

```
1     // Get LAN address Desktop
2     foreach (NetworkInterface item in NetworkInterface.
    GetAllNetworkInterfaces()) {
3         ...
4         allLocalIP.Add(ip.Address.ToString());
5     }
6     // Get LAN address Hololens
7 #if ENABLE_WINMD_SUPPORT && UNITY_WSA
8         foreach(HostName lhn in NetworkInformation.GetHostNames()){
9             ...
10            allLocalIP.Add(lhn.ToString());
11        }
12 #endif
```

Listing 5.17: `NetworkInterface` difference between desktop and Hololens implementation

5.3.4 Marker-base recognition

One of the best engine used for image-target recognition is the *Vuforia*. It is well integrated with the MRTK and exploits the spatial mapping features of the *Hololens* thanks to the extend tracking feature. So, an object remains on the located placed even if the camera does not frame the target.

Figure 5.18: *Vuforia*'s main scene elements on the Unity scene

The scene objects necessary to use the *Vuforia* engine are only two:

- the AR camera, which includes a `Vuforia Behaviour` component used to specify where the world reference is attached. In the *Hololens* case, it is always on the device itself. The behavior is defined by the `Vuforia Configuration` when the user sets up the device target, the image's database, extended tracking and many more options.

- the `ImageTargat` object which includes the `Image target behavior` component. It changes size depending on the marker's dimension, and the user can choose between the images inside the default database or from a custom one. The 3D model, that appear after the marker is tracked, should be its child on the scene's hierarchy. The user has all the freedom to positioning and scaling the object.

5.3.5 Sensors customization

The sensor's customization was completely redesigned for the *Hololens* implementation. As for the desktop implementation, it permits to add, remove, or update a sensor's upon the 3D model's surface, but there are executed only in `Menu` mode. When the image is tracked for the first time, the application performs some operation on the 3D model, similar to the one applied during the `Import` mode on the desktop application. All the described operation are implemented on the `MonitoredObject` class. In the `Awake` method, all the mesh components are listed and merged in a single one. In this case, it is essential that the model maintains the same position, size, and rotation concerning the root element (`ImageTarget`). Before the merging, the parent, position, rotation and scale's parameters of the `MonitoredObjetct` are saved in temporary variables, and then the `Transform` component reset. As in the desktop application, the delicate steps is to extract all the sensor's configuration data and position them correctly on the 3D model's surface. The application is performed to ensure that the sensor's size is not altered after the `MonitoredObjetct`'s `Transform` parameters are reinserted.

After that phase, the user can interact with the 3D object using the air-tap gesture thanks to the implemented `IMixedRealityPointerHandler`. Here the object reacts

differently when the user index is up, down or performs a "click" (rapid succession of up and down) and based on the application's status. Because the `Monitoring` implementation doesn't diverge from the default one, the focus is directed only on the `Menu` one.

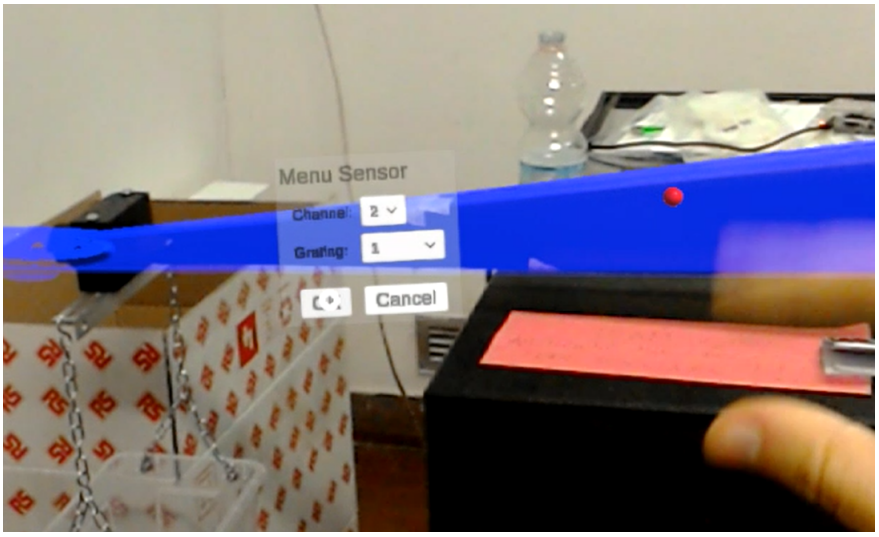


Figure 5.19: *Menu sensor* GUI for sensor's insertion

By performing an air-tap on the 3D model's surface, the application shows a new GUI. The main one is temporarily disabled to avoiding error or confusion by the user. The new `Canvas`, called *Menu Sensor*, permits the insertion of a new sensor that is not currently active on the 3D system. *Menu Sensor* is provided with two drop-down menus (*Channel* and *Sensors*) and two buttons (*Ok* and *Cancel*). The menu is locked on the pointed user's position and it is always facing the camera to be always visible by the user. The two drop-down menus show the available sensor depending on the selected channel's value. When its value is modified, the sensor's options are refreshed to contains only the available gratings. The chosen pair can be confirmed or cancelled the operation by pushing the namesake button. In both cases, the main GUI returns visible and *Menu Sensor* is disabled. Its behavior is managed by the `SensorCanvas` class which shows only the available sensor's for channel and forwards the user's decision to the `GameManager`. It updates the sensor's configuration, if necessary, and notifies the `MonitoredObject` to disable the *Menu Sensor*. The `MonitoredObject` saves the coordinate of the hit location and the associated normal. Those are necessary to place a sensor in the designated position and be sure that is rotated correctly with respect of the 3D model curvature's surface. The `SetSensor` method has some slight differences compared to the standard one. It discriminates the updates from the network from the other ones to avoid that the active sensors are repositioned in a wrong spot.
If the user points to a sensor's model, the *Menu Sensor* shows two different buttons, the *Update* and *Delete* one. The first substitutes the selected sensor with the one specified by the drop-down menus when the second deactivates it. In case the measurement type is `Displacement`, there is an additional option, *Rotate*. If pressed,

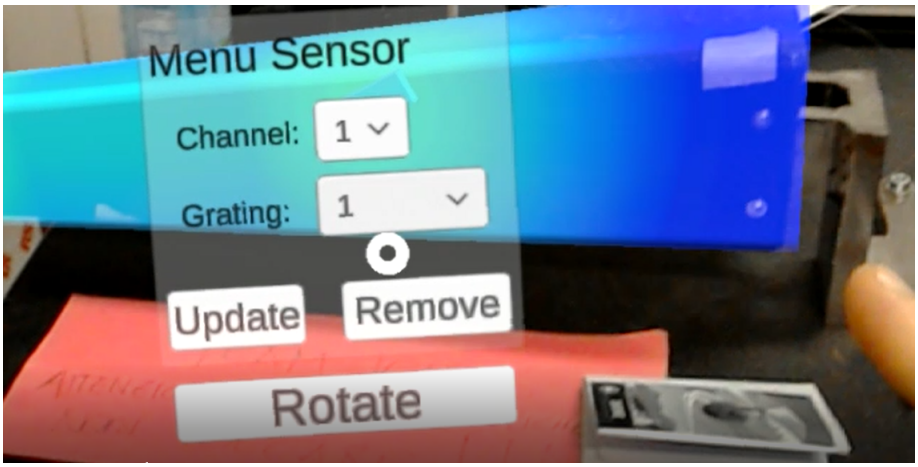


Figure 5.20: *Menu sensor* GUI for displacement sensor's customization

the user has to point the same sensor again and holds the pressed pose until the object reached the desired rotation. In this case, the *gaze* should pass through the sensor's model to be sure it works. The same rule of the desktop application is applied to rotate the object correctly.

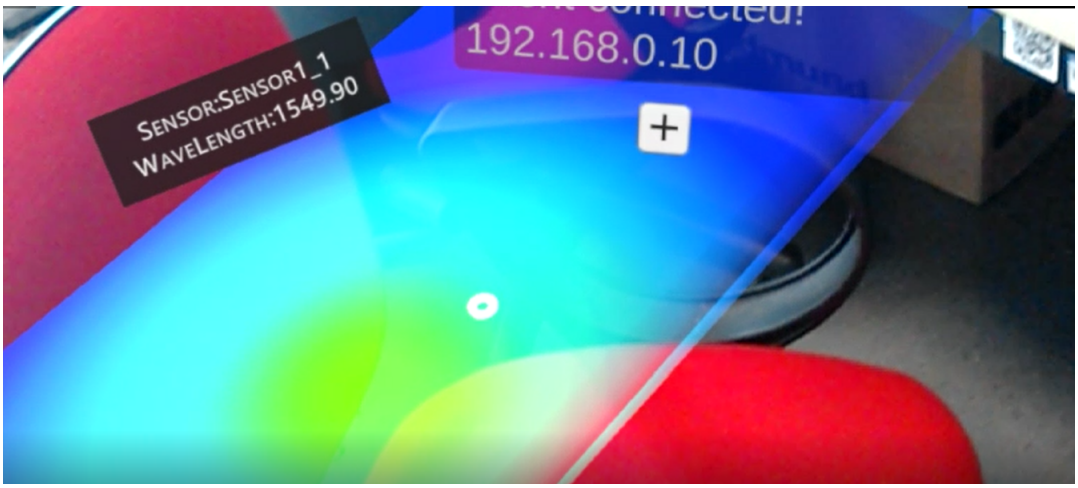


Figure 5.21: `Monitoring` phase example on the ICARUS wing

Chapter 6

Test and Result

The chapter presents the analysis of the proposed framework in terms of performance and usability. Each application is tested separately but using the same criteria. In term of performance, the most important features that need to be analyzed is the delay between when the middleware client sends the packet and when the visualization system updates the heatmap. Another important aspect is the utilization of CPU, GPU and memory. Unfortunately only one time it was possible to the *Hololens* on a real system environment due to technical problems, the others were performed with the desktop application.
Both are analyzed in different scenarios to determine the advantages and drawback of the framework.

6.1 Measurement Log

The delay analysis is performed thanks to an additional log file created only for this purpose. It measures the difference between the timestamp retrieved on the middleware client before it writes on the socket and the timestamp retrieved when the dispatcher calls the `UpdateShader` method. The two systems are synchronized on the same NTP server to reduce the time differences. The *Hololens* system was provided with an additional thread that transmits the data to another device in charge to save the information into a CSV file.

6.2 Performance: Case studies

The framework was tested in different cases scenarios using real physical systems or the interrogator emulator developed in [5]. The middleware client and the emulator itself were installed inside a *Raspberry Pie© 3 model B* on a *Rasbian* (Debian based distribution made for ARM processor series) OS. They are connected to a hidden WLAN without internet access to recreate an environment in which the whole architecture should be used. All of the tests are performed with the option *ShowAll panels* activate.

The physical tests case were executed on an laptop *Asus K53SD* with an *Intel©*

i5 2450M processor, a *NVIDIA GeForce 610M* and 8GB of RAM. Instead, the emulator tests were performed on a desktop machine with an *Intel© i7-7700 processor* with the integrated graphics card *Intel© HD 630* and 16 GB RAM.

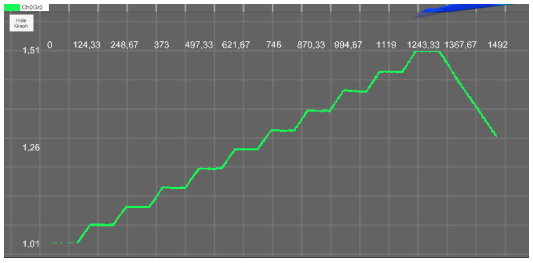
6.2.1 Case study: Carbon Fiber Reinforced Polymer sheet

This experiment consists of a bending test of a CFRP sheet where each step there is a delay of one minute (in this case only ten seconds). After it reaches the maximum bending, it slowly returns to its idle position. The configuration of the system had only one fiber active (channel 1, grating 1). The measure was performed only on the desktop application. The following table shows the average delay between all the simulation's period. Each sample represents a new `DataPacket` received inside the network stream.

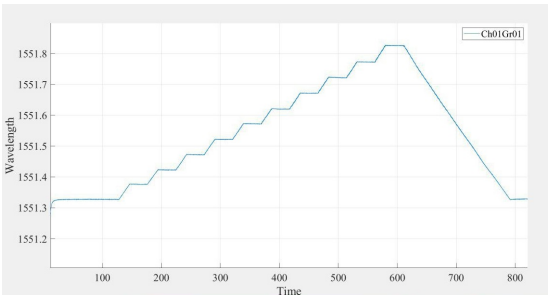
Simulation period	Number of packets	Average Delay
50s	6084	106,2ms
3min 01s	11940	102,8ms

Table 6.1: CFRP sheet measured delay in ms

At the end of the simulation, the application usage of CPU was 40% and the GPU 25%, in terms of memory it occupied 243MB. The application run smoothly without delay when interacted. The following graphs show consistence between the data received from the interrogator (calculated in *Matlab*) and the one visualized by the desktop application.



(a) Visualization framework real-time line graph



(b) *Matlab* line graph

Figure 6.1: Comparisons between line graph obtained from the sample received from the interrogator

6.2.2 Case study: Fuselage ICARUS

The most interesting experiment was performed on the fuselage of the *ICARUS* Unmanned Aerial Vehicle, developed by DIMEAS. Due to limited supplies of FBG fibers, the application was ultimately tested on the fuselage of the plane. Some initial bug-test was performed with the wing but no measurements were performed

at that time, but the application already showed great performance in terms of speed.

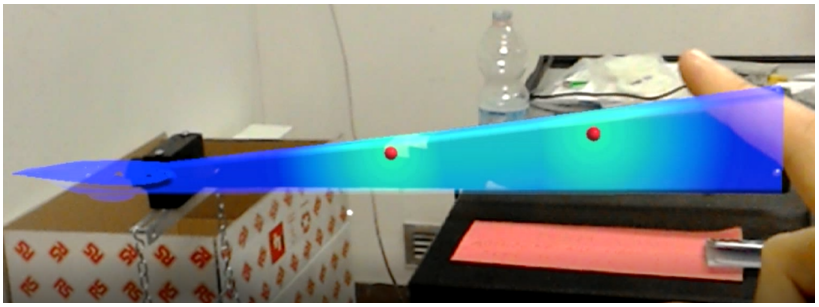


Figure 6.2: Fuselage displacement *Hololens* test

Figure 6.3: *ICARUS*'s first wing test

On the final part of the fuselage was attached a small plastic cart, water was gradually added to measure where and how it bends. To perform small tests it was used a set of weights instead. There are four fiber installed on the fuselage, with the following configuration: CH01GR01, CH02GR01, CH03GR01, CH03GR02. The test was performed on both the application, desktop and *Hololens*.

The following tables shows the average delay between all the simulation's period. Each sample represents a new `DataPacket` received inside the network stream.

Simulation period	Number of packets	Average Delay
26min 27s	94642	110,6ms

Table 6.2: Desktop: Fuselage displacement measured delay in ms

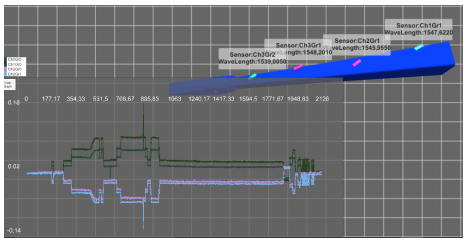
Simulation period	Number of packets	Average Delay
24min 20s	600249	74ms

Table 6.3: *Hololens*: Fuselage displacement measured delay in ms

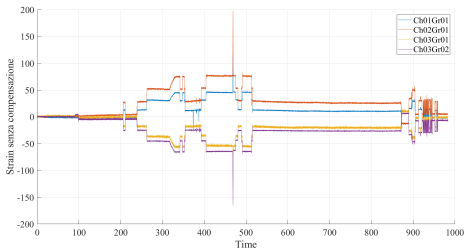
Similar to the previous test case, the desktop application usage of CPU this time was 50% and the GPU remains the same (25%). In this case, the memory usage

was a little bit higher reaching 300MB in total. Overall, it run smoothly for all the simulation period.
The *Hololens* showed, instead, high usage of the CPU, near the 100% with a variable refresh rate between 20 and 40 fps. The memory usage reached its peak at 250MB with a maximum of 900MB available. Even if the fps weren't constant, it run the application without any visual problem or slowdown.

The following graphs show consistence between the data received from the interrogator (calculated in *Matlab*) and the one visualized by the desktop application.



(a) Visualization framework real-time line graph



(b) *Matlab* line graph

Figure 6.4: Comparisons between line graph obtained from the sample received from the interrogator

6.2.3 Case study: Emulator

The biggest amount of tests were performed using the interrogator emulator due to the limited access to the *SmartScan*© interrogator. Besides, it was the only way to test the performance of the frameworks on its maximum capacity or, in general with a high number of active sensors.
The desktop and *Hololens* application were tested with different sensors' configuration, which vary in duration and sensors' number. In this way it is possible to see how both react in different scenarios. On the desktop environment, some of the monitored logs required data cleaning because when the application is set to background it is paused and the packets are not processed until it is reset as foreground. Due to this issue, some of the timespans were altered and so not considered.

# Sensors	Simulation period	Number of packets	Average Delay
4	18min 2s	130724	66,2ms
12	2min 12s	13464	69,5ms
64	52min 4s	380954	94.4ms

Table 6.4: Desktop: Emulator delay in ms

The *Hololens* showed, even in those cases, high usage of the CPU, near the 100%, and memory usage with a peak of 248,1MB. It maintained an acceptable framerate on the first three scenarios, with values between 17 and 34 fps, but, in the latter situation, they dropped on a fix 7fps for all the monitoring phase. Due to

# Sensors	Simulation period	Number of packets	Average Delay
2	6min 20s	199140	50,6ms
8	5min 24s	141579	80,4ms
20	1min 17s	14534	81,0ms
40	9min 14s	215608	99.2ms

Table 6.5: *Hololens*: Emulator delay in ms

those result, it possible assume that the application runs very poorly with all the 64 active sensors . The motivations could be various, but hardware limitations, poor optimization and the speed in which the device has to update all the scene elements could be the main causes of this phenomena. It could be interesting to test the application on the announced *Hololens 2* to verify these observations.

Furthermore, on the desktop application is tested how the presence of the real-time graph affects the performance in term of CPU, GPU and memory usage. The two test case were chosen base on the feature of the graph to be scrollable after it reaches a number of points per line. The tests are performed in the same timespan of 30 minutes to obtains a fair comparison.

# Sensors	Graph	CPU %	GPU %	Memory(MB)
8	Yes	22	59	189.4
8	No	22.5	77,6	182.8
64	Yes	30.78	54.2	227.5
64	No	21.2	61.6	208.9

Table 6.6: Desktop: performance with and without the real-time graph

6.2.4 Case studies analysis

Overall the framework performs with quite well based on the provided requirements with an average delay of 100ms for each packet. In particular, the desktop results showed a slight delay during both physical tests if compared with the emulator. It can be caused by the limited hardware capability of the laptop, which was released in 2011 when the hardware components of the desktop were on the market since 2017. The observation is also confirmed by the *Hololens* outcomes, which didn't show any relevant differences between the various test cases.
The *Hololens* showed difficulties to manage the rendering, especially in configuration with more than 40 sensors. With the option *ShowAll* turned off, it manages the configuration without a similar performance of the 20 sensors' one. The only test performed with 56 sensors shows similar fps results of the previous case, but when the *ShowAll* checkbox is activated the application quickly slow down until it is not able to render the scene anymore. Based on the data collected by the external devices, the application was, in fact, receiving and analyzing the packets but wasn't able to update all the scene objects for the scheduled time, even if the majority of

it was only text. As said on the previous section, this can be due to the *Hololens* limited hardware capability, which is enforced by the fact that, in general, the CPU usage of the application was always near the 100% with small reduction when only the `Canvas` was framed.

The desktop application shows different performance based on the hardware in which was executed. In general, memory usage showed good results in both cases, but CPU and GPU vary depending on the hardware components. As said before, the CPU usage variation can be justified by the outdated hardware of the laptop. An interesting observation can be made regarding the GPU usage. The laptop reported a shallow usage compared to the desktop, which was quite high. It can be caused by a lack of external GPU on the desktop machine when in the laptop one is assembled with an *Nvidia* one. Even if it is quite outdated, it shows better performance compared to the integrated card. In addition, the application was tested on another machine with an *Intel© i7-6700* processor and a *Nvidia Geforce 1060 6GB*, and it showed similar CPU e Memory usage but a lower usage of the GPU with only 10%. Besides, the desktop application shows no relevant performance variation between the visualization or not of the line graph.

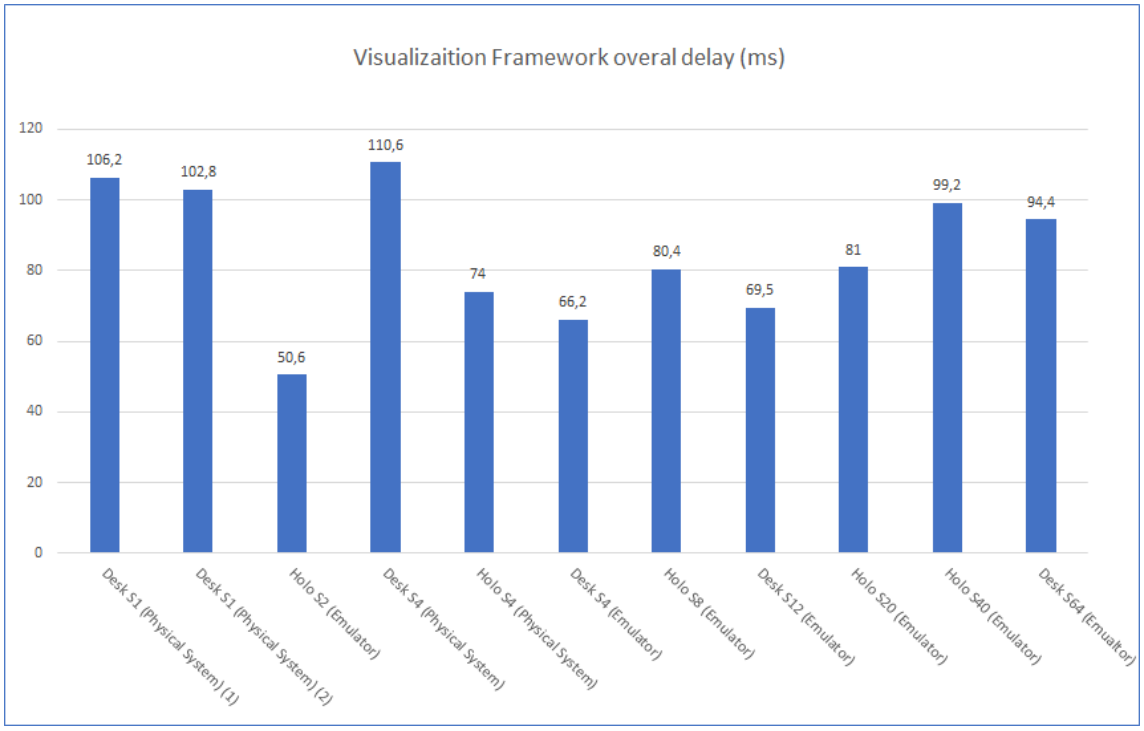


Figure 6.5: Visualization Framework Delay overview

6.3 Review and usability test

Another important review aspect was the usability of the framework. It was decided to make the Icarus's member test both applications and then ask a few questions.

Because of the limited number of people that had to work with the FBG fibres, it wasn't possible to perform a test with a huge number of people but it was considered more relevant show the application to real end-users instead test it with a general audience due to the circumstance in which this application should be used.

The desktop application results easy and immediate to use after it was explained the principal commands. The existence of a customizable heatmap, model importer and the pop-up panels was really appreciated. However, the real-time graph was the feature that received the biggest praise, with a limited interest in the "visualization" part of the framework. The *Hololens*, instead, manifested a huge general enthusiasm. Even if the first minutes there was a little bit of confusion due to the unfamiliarity with this kind of devices, the users didn't have any problem to learn the commands and, as for the desktop one, it was intuitive and straightforward to use. However, they reported feeling a little bit annoyed when using the virtual keyboard. In the end, they seem interested in using the application on a flight test to monitoring purposes.

The framework was presented to the *PhotoNext* (Inter-Dipartimental Center for Photonic technologies) research group and different professors of DIMEAS and DISAT interested in the product. A demo of the desktop application was performed to show in real time its performance, plus it was showed a demonstrative video of the *Hololens* part, recorded when tested on the ICARUS's fuselage. At the end, they were satisfied by the final results, so much that the whole framework will be exhibited at the *Festival della Tecnologia* in November 2019 in Turin.

Chapter 7

Future Work

In the following chapter lists all the possible improvements and new features that can be included in the mixed real-time visualization frameworks in future releases.

7.1 Missing features

Even if the platform is entirely functional, it requires some fixes and improvement for reaching its full potential. The *Hololens* port doesn't support a discrete number of features available on the desktop one. It could be interesting to implement a full custom system when the user can choose from the cloud platform the 3D model that it wants to use during the simulation. Besides, it can support a configuration system that preserves all the necessary parameters to position the monitored object in the correct position given the marker position. In this way, the application doesn't have to worry about the format of the 3D model, because the cloud platform provides to send the correct data. A similar feature can be extended to the desktop application, in particular, it could be interesting to increase the 3D formats and add use a configuration system in which the user can save its setting such network address, sensors' position, default heatmap and other custom parameters.

7.2 Custom image-tracking and real-time graph implementation

The initial framework was developed to be released as an open source tool that anyone can use, test and improve. Unfortunately, both the mark-base system (*Vuforia*) and the real-time graph (*Graph and Chart* [23]) are not open source tools. *Vuforia* gives a *Free Development license* for developing and prototyping but since June 2019 its required a paid license to sell or distribute the product. A similar observation can be made for the Unity plug-in because it is can only be purchased on the Unity Assets store, so doesn't exist an open source release at the time. An alternative of the mark-base system it could be the usage of a Machine Learning model to perform object recognition and pose estimation. For example, the services provided by the

Azure platform allow to create and train a simple model to perform Object recognition that is compatible with the *Hololens*'s device. In addition, *Microsoft©* releases in June an article in which explains how developers can exploit the functionality of the *Hololens* camera and so developers can implement a custom object recognition system exploits open source library such as *OpenCV*.
Graph and Chart, even if its the best tool on the market, it cannot be applied to the context of an open source platform. The plug-in was used to reduce the developing time and use a tool already well know and tested by the Unity developers. Overall, it could be interesting to implement a simplified version of the tool that works only as a real-time line graph. It can require a long developing time due to the performance requirements, but for the framework, it can be highly beneficial because it can be implemented to make possible a port of this tool for the *Hololens*' implementation.

7.3 Hololens 2 porting

Thanks to the MRTK V2.0 RC1, the developed code for *Hololens* application should be already compatible for the announced *Hololens 2*. However, due to the state of the SDK, the first official released is not published yet so it could be plausible that some minor changes occur. Even with small modifications, it should be a priority to test the application on the new hardware as soon as it is distributed. According to the MRTK, the new features should resolve the main issues that the current application has. For example, the virtual keyboard usage should become more intuitive to use because the device will be able to understand the hand's direction and point an object with them instead of using the gaze. Also, the integration of an *ARM©* processor (*Qualcomm Snapdragon 850*) instead of using an *Intel© Atom* will improve performance and power efficiency. Especially in the application's case which suffers from a low fps rate.

Chapter 8

Conclusion

This thesis aims to produce, develop and test a Mixed real-time visualization framework for FBG IoT sensors. The expected goal was to improve the previous IoT architecture for FBG sensors with a system that manages the monitoring of a generic physical object by exploiting the VR and AR technologies. The choice of using the *Microsoft© Hololens* headset provides to thought carefully about the structure of the framework. After an in-depth analysis of the previous work and the requirements, the system had to reach a good compromise between speed and performance. It was designed a set of core features and classes that can easily be exploited in different platform and can follow a general structure that obtains solid performances with minimal delays between the original architecture and the visualization system. Then, the two application (desktop and *Hololens*), which composed the framework, required just small adjustment to the core structure. However, the two platforms had their weak and strong points that needed different design choice to an efficient and comfortable experience for the end user.

The project is a collaboration with DIMEAS, DISAT and DAUIN departments as part of the Inter-Departmental Center for Photonic technologies (*PhotoNext*). It results in a stimulating environment that makes the design process quite challenging. In particular, the collaboration with the *ICARUS* team's member was crucial for the final product. Thanks to them, it was possible to discuss and decides the essential features that need to be included inside the framework and how they should be presented to offers a user-friendly experience.

The final result showed good performance, which respected the initial requirements, in real and simulated case scenario. Both interfaces were considered easy to understand, and the different features pretty straightforward to learn and remember.

Acronyms

API	Application Programming Interface
AR	Augmented Reality
ARIoT	Augmented Reality and Internet of Things
CAD	Computer-Aided Design
CFRP	Carbon Fiber Reinforced Polymer
CG	C for Graphics
CSV	Comma-separated values
CPU	Central Processing Unit
eMMC	embedded MultiMediaCard
FBG	Fibre Bragg Grating
DAUIN	Dipartimento di Automatica e Informatica
DIMEAS	Dipartimento di Ingegneria Meccanica e Aerospaziale
DISAT	Dipartimento Scienza Applicata e Tecnologia
DLL	Dynamic-Link Library
FOV	Field Of View
GIS	Geographic Information System
GLSL	OpenGL Shading Language
GNSS	Global Navigation Satellite System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HHD	Hand-Held Display
HLSL	High Level Shader Language

HMD Head Mounted Display

HPU Holographic Processing Unit

IAR Industrial Augmented Reality

IL2CPP Intermediate Language To C++

IMU Inertial Measurement Unit

IoT Internet of Things

JPEG Joint Photographic Experts Group

JSON JavaScript Object Notation

LPDDR Low-Power Double Data Rate Synchronous Dynamic Random Access Memory

MTG Model Target Generator

MRTK Microsft Mixed Reality Toolkit

NTP Network Time Protocol

OOP Object Oriented Programming

RGB Red Green Blue

RC Release Candidate

SDK Software Development Kit

SIGGRAPH Special Interest Group on GRAPHics and Interactive Techniques

SLR Systematic Literature Review

SoC System on Chip

SRAM Static Random-Access Memory

PNG Portable Network Graphics

TAP Task-based Asynchronous Pattern

TCP Transmission Control Protocol

TLS Transport Layer Security

UI User Interface

UDP User Datagram Protocol

UWP Universal Windows Platform

VR Virtual Reality

WR Windows Runtime

WLAN Wireless local area network

Bibliography

[1] T. T. Amici, P. H. Filho, and A. B. Campo. Augmented reality applied to a wireless power measurement system of an industrial 4.0 advanced manufacturing line. In *2018 13th IEEE International Conference on Industry Applications (INDUSCON)*, 2018.

[2] V. Cozzolino, O. Moroz, and A. Y. Ding. The virtual factory: Hologram-enabled control and monitoring of industrial IoT devices. In *2018 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*, 2018.

[3] Edison Pignaton de Freitas Diogo Augusto Pereira, Wagner Ourique de Morais. NoSQL real-time database performance comparison, 2017.

[4] P. Fraga-Lamas, T. M. Fernández-Caramés, Ó Blanco-Novoa, and M. A. Vilar-Montesinos. A review on industrial augmented reality systems for the industry 4.0 shipyard. *IEEE Access*, 2018.

[5] Mauro Guerrera. Algorithms and methods for fiber bragg gratings sensor networks. Master's thesis, Politecnico di Torino, 2018.

[6] A. Z. Abdul Halim. Applications of augmented reality for inspection and maintenance process in automotive industry. In *Journal of Fundamental and Applied Sciences*, 2018.

[7] R. Hamidane, L. H. MOUSS, A. Bellarbi, and R. MAHDAOUI. Implementation of a preventive maintenance system based on augmented reality. In *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*, 2018.

[8] PTC Inc. Vuforia developer library. `https://library.vuforia.com`.

[9] Bitsplash Interactive. Graph and chart documentation. `http://bitsplash.io/graph-and-chart`.

[10] D. Krupke, F. Steinicke, P. Lubos, Y. Jonetzko, M. Görner, and J. Zhang. Comparison of multimodal heading and pointing gestures for co-located mixed reality human-robot interaction. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018.

[11] Q. Liu, Q. Wei, J. Luo, Z. Li, Z. Zhou, and Y. Fang. A bi-mode state monitoring system for aero-engine components based on FBG sensing. In *Proceedings of the 2014 International Conference on Innovative Design and Manufacturing (ICIDM)*, 2014.

[12] Microsoft. Gestured. `http://archive.is/v8Two`.

[13] Microsoft. Mixed reality documentation. `https://docs.microsoft.com/it-it/windows/mixed-reality/`.

[14] Geoffrey Momin, Raj Panchal, Daniel Liu, and Sharman Perera. Case study: Enhancing human reliability with artificial intelligence and augmented reality tools for nuclear maintenance. In *Volume 2: Heat Exchanger Technologies; Plant Performance; Thermal Hydraulics and Computational Fluid Dynamics; Water Management for Power Systems; Student Competition*, 2018.

[15] Francesco De Pace, Federico Manuri, and Andrea Sanna. Augmented reality in industry 4.0. *American Journal of Computer Science and Information Technology*, 2018.

[16] Riccardo Palmarini, John Ahmet Erkoyuncu, Rajkumar Roy, and Hosein Torabmostaedi. A systematic review of augmented reality applications in maintenance. *Robotics and Computer-Integrated Manufacturing*, 2018.

[17] Photonext. Photonext. `http://www.photonext.polito.it/it/`.

[18] PimDeWitte. Unitymainthreaddispatcher. `https://github.com/PimDeWitte/UnityMainThreadDispatcher`.

[19] F. Pires, J. Barbosa, and P. Leitão. Quo vadis industry 4.0: An overview based on scientific publications analytics. In *2018 IEEE 27th International Symposium on Industrial Electronics (ISIE)*, 2018.

[20] D. Qiu and L. Gao. Application of virtual reality technology in bridge structure safety monitoring. In *2010 International Conference on Computer and Information Application*, 2010.

[21] V. Rajan, N. V. Sobhana, and R. Jayakrishnan. Machine fault diagnostics and condition monitoring using augmented reality and IoT. In *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*, 2018.

[22] Daniel Segovia, Miguel Mendoza, Eloy Mendoza, and Eduardo González. Augmented reality as a tool for production and quality monitoring. *Procedia Computer Science*, 2015.

[23] Unity Access Store. Graph and chart by bitsplash interactive. `https://assetstore.unity.com/packages/tools/gui/graph-and-chart-78488`.

[24] Unity Access Store. Runtime obj importer by dummiesman. `https://assetstore.unity.com/packages/tools/modeling/runtime-obj-importer-49547`.

[25] Unity Access Store. Simple file browser by graces games. `https://assetstore.unity.com/packages/tools/input-management/simple-file-browser-98451`.

[26] Mixed Reality Toolkit Team. What is the mixed reality toolkit. `https://microsoft.github.io/MixedRealityToolkit-Unity/README.html`.

[27] Infibra Technologies. Fbg overview. `http://www.infibratechnologies.com/technologies/fiber-bragg-gratings.html`.

[28] Unity Technologies. Tutorial - rendering and shading. `https://learn.unity.com/tutorial/rendering-and-shading#5c7f8528edbc2a002053b538`.

[29] Unity Technologies. Unity api documentation. `https://docs.unity3d.com/`.

[30] Unity Technologies. Unity api documentation. `https://unity3d.com/unity/features/multiplatform/vr-ar`.

[31] Unity Technologies. What is a game engine? `https://unity3d.com/`

what-is-a-game-engine.

[32] Reid Vassallo, Adam Rankin, Elvis C. S. Chen, and Terry M. Peters. Hologram stability evaluation for microsoft HoloLens. In *Medical Imaging 2017: Image Perception, Observer Performance, and Technology Assessment*, 2017.

[33] O. Wasenmüller, M. Meyer, and D. Stricker. Augmented reality 3d discrepancy check in industrial applications. In *2016 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, 2016.

[34] S. Webel, M. Becker, D. Stricker, and H. Wuest. Identifying differences between CAD and physical mock-ups using AR. In *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, 2007.

[35] I. Wijesooriya, D. Wijewardana, T. De Silva, and C. Gamage. Demo abstract: Enhanced real-time machine inspection with mobile augmented reality for maintenance and repair. In *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 2017.

[36] Wikipedia. Microsoft hololens. https://en.wikipedia.org/wiki/Microsoft_HoloLens.

[37] Jun Xu Xiyang XU Xiongming Zhou, Wenqun Su. 3d real-time display system for cable temperature monitoring. In *CICED 2010 Proceedings*, 2010.

[38] Alan Zucconi. Arrays and shaders: heatmaps in unity. https://www.alanzucconi.com/2016/01/27/arrays-shaders-heatmaps-in-unity3d/.