POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering

Master's Degree Thesis

Augmented Reality for the Visualization of Flight Information Coming From a Model Aircraft

Supervisors
Prof. Bartolomeo Montrucchio
Dr. Antonio Costantino Marceddu

Candidate
Luca Viscanti

April 2022

Summary

The *PhotoNext* project consists in retrieving and processing information coming from a set of Fibre Bragg Grating (FBG) sensors, so that its status can be visualized in a simple and immediate way through a heat-mapped 3D model. It is carried out in collaboration with the Department of Control and Computer Engineering (DAUIN) and the Department of Mechanical and Aerospace Engineering (DIMEAS) of the Politecnico di Torino, as a part of the Inter-Departmental Center for Photonic technologies. The entire architecture is primarily developed to analyze and monitor a model aircraft created by the *Icarus* team in Politecnico di Torino.

FBG sensors are devices that exploits photosensitivity: they measure the variation of the refractive index of the optical fiber core caused by some physical effect such as strain, pressure and temperature. This technology is particularly used as part of a preventive monitoring system. The flight information are displayed through an application designed for Augmented Reality using *Microsoft© HoloLens 2* headset called *HoloLens Viewer*. This allow the user to have an overview of both the aircraft during flight but also of the digital information coming from the sensors for real-time monitoring, or offline with an old collection retrieved from the database to analyze a past session.

The aim of this thesis is to design an improved visualization framework exclusively for the *HoloLens 2* device, in order to unify the two existing versions in a single enhanced version that can be used in real-time during a flight session. During this monitoring session the program shows information in two different ways. The first give an immediate visual feedback in the 3D model: for each placed sensor there are heat-maps that show the sensor data in real time, with different colors based on its intensity and type. The second is a more technical view: a graph is shown with the wavelength values for each sensor over time, updated in real-time to allow the pilot or co-pilot to analyze the flight data trend.

Since the app should work regardless of the type of model in which the sensors are placed, it is possible to import different 3D models, and to increase the ease of use it is possible to save a configuration where the sensors are placed in the model to reuse it later. A configuration file with some information is stored in the memory: HoloLens Viewer will read the contents of this file to automatically set some parameters, so that the user can avoid manually setting his system data while

wearing the headset and immediately activate the display. The network connection of the application can be done via database or TCP protocol. In the latter case, it communicates directly with the middleware in charge of retrieving the data from the sensors.

The final tests of the HoloLens Viewer monitored the usage of the CPU, GPU, memory and the number of frames per second. Several scenarios were analyzed. Using a wing and the tail of a model aircraft, created by the *Icarus* team, it was possible to analyze the real behavior, although tests were performed in the laboratory and not during a flight. Other analyzes were performed using a software that emulates the behavior of FBG sensors, which was created with the aim of simplifying test procedure. HoloLens Viewer showed nice results in term of performance. It was also tested by some members of the *Icarus* team, receiving satisfactory feedback.

Acknowledgements

First of all, I would like to thank prof. Bartolomeo Montrucchio and dr. Antonio Costantino Marceddu for their support during the realization of this thesis.

A special thanks to my mother Carla and my father Giacomo, who have always assisted me throughout my life sharing joys, sacrifices and successes.

Finally, I would like to thank my other family members, friends, and colleagues who have supported me along the way.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

One of the direct evolution from the use of the Internet network is represented by the development of the concept of the Internet of Things (IoT). In this context, physical objects are embedded with sensors and have processing ability, making themselves recognizable and intelligent as they can share information with other devices. This field is constantly evolving thanks to the convergence of various technologies, such as increasingly powerful embedded systems, machine learning and real-time analysis.

In a monitoring environment, an IoT system collects a very large amount of data to be deposited in a massive database, and subsequently to be displayed and analyzed. From this basis arises the need, by the *PhotoNext* research group of the Politecnico di Torino, to create an architecture that can collect data from sensors of a specific type, called Fiber Bragg Grating (FBG), to process and show their information content in real-time.

FBG sensors exploit photosensitivity to measure changes in the refractive index of an optical fiber caused by physical effects such as strain, pressure and temperature. Due to their properties, they can be mounted in a model aircraft, to be used as part of a preventive monitoring system during a flight session. In this particular case, the sensors can detect the bending of structural elements of the aircraft and the temperature in specific positions. In general, to view the data of the FBG sensors, it is necessary to use software generally developed by the same vendor of the hardware capable of reading the values from the sensors. This hardware is usually called interrogator. To the best of our knowledge, no other open-source real-time visualization systems seem to exist. Furthermore, in literature was not found any software able to show in real-time the position of the FBG sensors and their values in a 3D model. For these reasons, real-time monitoring framework has been developed.

The framework architecture is divided into various parts:

- a dedicated hardware system that detects raw data from sensors mounted on the aircraft, called interrogator;
- a software capable of collecting the data coming from the interrogator, processing and sending them to a cloud database, called middleware;
- an application software for the end-user that allows the visualization of this data in real-time and not.

This thesis aims to improve the architecture of the PhotoNext project, developing a new application exclusively for the *Microsoft© HoloLens 2* Augmented Reality headset for displaying sensor information. It is meant for in-site monitoring, so it can be used by a pilot or co-pilot to analyze the flight trend.

The application, called HoloLens Viewer, allows viewing FBG sensor data in two different ways. As introduced before, the end-user must be able to monitor the entire flight session; for this reason, a graph displays the wavelength values of each sensor over time and is updated in real-time. The second way of showing the information is more immediate: a 3D model with sensors visible on the surface of the plane reacts according to their intensity value by coloring a radial portion of the surface, making the model more interactive. Saving the data in a database allows viewing the data even in non-real-time mode, similar to the live version. Thanks to Augmented Reality, the user is free to move around and interact with it by using hand gestures: one of the strong point of the application is that, by recurring to them, it makes possible to change the size and orientation of the model and the position of the sensors on its surface in a very immersive way. Since the app should work regardless of the type of model in which the sensors are placed, it is possible to import different 3D models. It also offers the possibility to save a configuration where the sensors are placed in the model to reuse it later. A configuration file with some information is stored in the memory: each time it is opened, HoloLens Viewer will read the contents of this file to automatically set some parameters, so that the user can avoid manually setting his system data while wearing the headset and immediately activate the display.

The project was developed in collaboration with the Department of Control and Computer Engineering (DAUIN) and the Department of Mechanical and Aerospace Engineering (DIMEAS) of the Politecnico di Torino, as a part of the Inter-Departmental Center for Photonic technologies. The partnership with the Icarus team in Politecnico di Torino allowed to analyze and monitor a real physical system: an Unmanned Aerial Vehicle (UAV) called Anubi.

1.2 Augmented Reality

Augmented Reality (AR) means the enrichment of human sensory perception through virtual information that would not be perceptible with the five senses. AR can be defined as a system that realizes several characteristics together: a real-time interaction and a combination of the real and digital world, showing virtual objects in a real environment. One of the first examples of this technology dates back to 1968 thanks to Ivan Sutherland: he was the first to develop a head-mounted display. In the following years, it was tried to apply this technology in the industrial field. In this area, the first implementation documentation was presented in 1990, thanks to two Boeing engineers (Tom Caudell and David Mizell), who developed a head-mounted display to visualize aircraft specific schematics on a board [1].

Nowadays, the two commonly used tools for AR are:

- smartphones provided with a Global Positioning System (GPS), a magnetometer (compass) and with the possibility to display a video stream in real-time, as well as an Internet connection to receive online data.

- computers based on the use of markers. They can be recognized by the PC thanks to the webcam, and multimedia contents are superimposed on them in real-time.

In addition to these easier-to-use tools there are many technologies under development, including Microsoft's AR headset, called HoloLens, or other even more futuristic tools such as contact lenses that display digital images (still in a phase of development).

Figure 1.1: *Microsoft© HoloLens 2* headset [2].

Microsoft Hololens is mainly used in the industrial setting and allows creating more agile factories by reducing downtime. With this device, employees can quickly learn complex tasks and collaborate from anywhere.

1.3 Thesis structure

This thesis is divided into seven chapters. Chapter 2 examines previous works, giving a general description of the design of the system. Chapter 3 describes the design of the application created and its features. The used frameworks are examined in Chapter 4, while the description of their implementation in Chapter 5. Chapter 6 presents the tests performed and their outcomes. Finally, Chapter 7 contains some ideas for future development and the conclusions about the work made.

Chapter 2

Previous works

Before analysing the proposed and designed implementation, previous work will be described, giving a general description of the design of the system, focusing on each layer and its interconnections.

2.1 Overview

PhotoNext is an interdepartmental center for Applied Photonics launched by Politecnico di Torino during summer 2017. The group focus its activities on three main areas:

- ultra-high speed optical networks for next-generation ultra-broadband optical communications;
- innovative optical sensors for safety, industrial, civil and life science applications;
- optical components with innovative functionalities to generate, manipulate and detect light.

The architecture regarding FBG technology and system monitoring is a part of *PhotoNext* research group projects.

Figure 2.1: PhotoNext logo [3].

2.2 System architecture

The overall system architecture is designed to provide all required information to monitor a physical system from its sensors, and it is developed to create a common model independent of the physical system, the tools and the interrogator used.

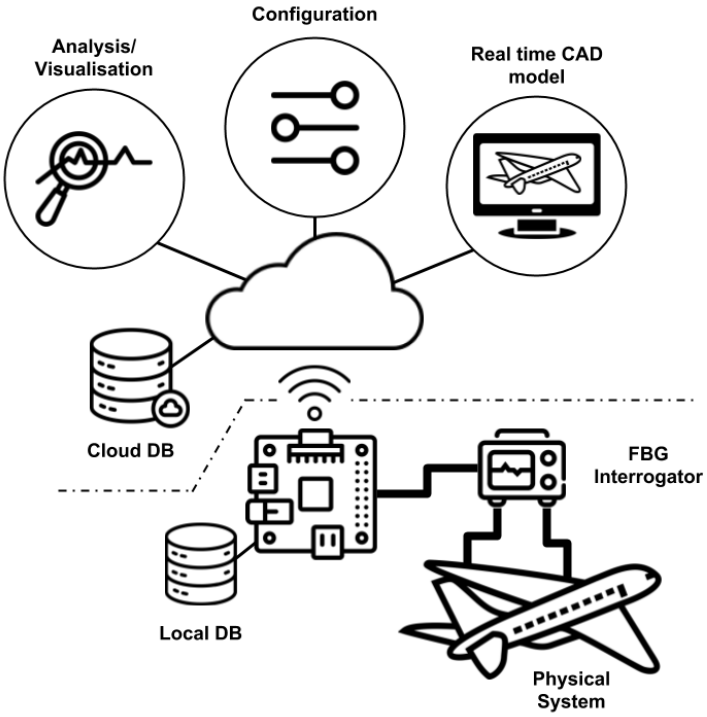


Figure 2.2: System architecture [4].

The system is composed of six main layers:

- **Physical system**: a set of FBG sensors and the model aircraft to be monitored;
- **Interrogator**: the hardware that reads the data from the sensors;
- **Emulator**: layer that replaces the interrogator, for developing purposes;
- **Middleware**: it extracts information from the interrogator and prepare data for the next layer;
- **Cloud network**: it stores data coming from the previous layer;
- **Visualization system**: the application that displays information in real-time to make analysis.

2.2.1 Physical system

The physical layer is composed of two different parts: one is the technology used to measure several physical parameters of the system, and the second is the real object to be analyzed, which can be any device that needs to be monitored.

Fibre Bragg Grating (FBG) technology

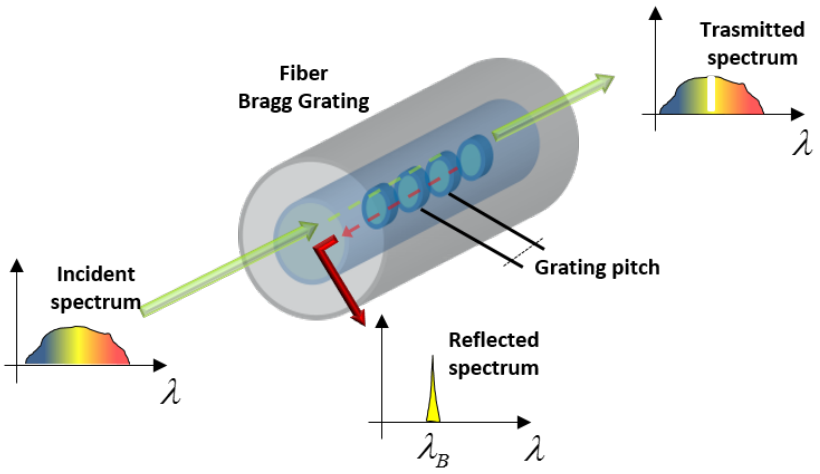


Figure 2.3: FBG sensor structure [4].

Fibre Bragg Grating technology makes possible to measure physical parameters such as temperature, strain, pressure and displacement. As the name implies, it is a device that exploits photosensitivity: in its simplest form, it is a periodic modulation of the permanent refractive index inscribed in the nucleus of the optical fiber. Sensors built with this technology are highly reliable, low-priced and less invasive than other sensors: for this reasons they are applied in various fields, from energy production plants to civil engineering, as structural monitoring.

The perturbation index in the core acts as a stop band filter. As the fiber is compressed, folded or stretched, the index value changes. The strongest interaction occurs at the Bragg wavelength Λ given by [5]

$$\lambda_B = 2 \cdot n_{eff}\Lambda \tag{2.1}$$

where n_{eff} is the effective core refractive index and Λ is the grating period. If there is any change in the microstructure, such as strain or temperature, the refractive index also change, and it can be calculated with the following formula:

$$\frac{\Delta\lambda}{\lambda_0} = \frac{\Delta(2n_{eff}\Lambda)}{2n_{eff}\Lambda} \tag{2.2}$$

Taking as an example the strain modification, 2.2 becomes as follows:

$$\frac{\Delta\lambda}{\lambda_0} = \frac{\Delta(2n_{eff}\Lambda)}{2n_{eff}\Lambda} = \varepsilon_1 - (\frac{n_{eff}^2}{2})[p_{11}\varepsilon_t + p_{12}(\varepsilon_1 + \varepsilon_t)] \tag{2.3}$$

where ε_1 are the principal strains along the fiber axis and ε_t is the transverse to the fiber axis [5]. If the strain is homogeneous and isotropic, then it can be simplified to its more common form:

$$\frac{\Delta\lambda}{\lambda_0} = [1 + p_e]\Delta\varepsilon \tag{2.4}$$

with p_e as the photoelastic constant [6]. The temperature sensitivity of a bare fiber is primarily due to the thermo-optic effect, and is given by:

$$\frac{\Delta\lambda}{\lambda_0} = (\alpha + \xi)\Delta T \tag{2.5}$$

with α as the coefficient of thermal expansion of the fiber and the thermo-optic coefficient ξ [6].

A separate tool, called interrogator, is used to extract data from the sensors.

Model aircraft

The analyzed system in this project is a Unmanned Aerial Vehicle (UAV) built by *Innovation Center for Amateur Rocketry and Unmanned Ships* (ICARUS) Team. FBG sensors are positioned throughout the aircraft structure, allowing the remote monitoring of temperature and displacement. The entire architecture can also be used to monitor systems which differs from aircrafts and that needs to identify real-time problems.

Figure 2.4: Model aircraft *Anubi*, from ICARUS [7].

2.2.2 Interrogator

The interrogator can detect and measure any changes in the reflective index of the fiber. It provides the raw data of the FBG sensors or some finer, called peak data. *SmartScan*© from *SmartFibres*© is the hardware chosen for this case study. It can read 64 different FBGs, with 4 channels and 16 gratings per channel, and it communicates with the *SmartSoft Application Software* using the LAN connection and a custom UDP protocol. The fundamental keys to this choice are its robustness, speed and great resolution.

It is connected to the middleware layer through an abstraction interface.

Figure 2.5: *SmartScan* interrogator [8].

2.2.3 Emulator

The difficulty of real-time testing with the interrogator and FGBs disclosed the need to create a custom developed *emulator* software capable of replicating the behavior of *SmartScan* device. It provides random values corresponding to the peak wavelength for each FGB sensor and it generates the same UDP traffic [9].

2.2.4 Middleware

The middleware is the part responsible for processing raw data coming at high speed from the interrogator in order to be usable by the next system stages. It provides an extra security layer establishing an encrypted connection to the data server. Therefore, it connects components and applications that are not designed to communicate with each other. It was implemented in C++ and runs on a *Raspberry*

Pi 3 Model B. It was designed to continuously receive packets from the interrogator, to parse them in a human-readable format and to finally send them to the cloud database, so it is inherently a multi-thread application. It is independent from the type of hardware interrogator, but since the previous layer can send too much data at high rate, the purpose of the software is to filter data, keeping the same level of details and maintaining a low server traffic. The software also adds some metadata to the associated information, like time-stamp and others needed for the visualization framework or for offline analysis. More detailed information about this software can be found in the works of El Zein [10] and Scaldaferri [6].

2.2.5 Cloud network

The data coming from the previous layer are stored in the cloud. The platform open source *KaaIoT* [11], frequently used in IoT applications, was initially chosen for this purpose. Unfortunately, the open-source version was deprecated years ago, so the cloud network architecture switched to a NoSQL database: *MongoDB*®.

MongoDB is a non-relational database designed to be more easily scalable than relational ones. The main difference with SQL database is that is based on collection and documents instead of tables, rows and columns. More details will be explained in Section 4.3.

2.2.6 Visualization system

The viewer is the end-point of the architecture, designed for those who has in charge to monitor the system. It must show in a very effective and simple way the data coming from FBG sensors and fine-tuned by the middleware to allow their real-time analysis while the physical system is working. This layer can show where sensors are placed within the physical system, mapping them onto a CAD model.

There are two different applications for this purpose: a desktop version and a *HoloLens* version for the first device developed by *Microsoft*©, which share the same basic structure, with some differences inherently due to the hardware limitations of the second. This thesis aims to create a new improved application exclusively for the brand new *Hoolens 2* device, in order to unify the two versions in a single enhanced version. It can be used in real-time during a flight session, allowing the user to see directly the actual model aircraft, and at the same time see a holographic version of it with real-time flight data. A brief description and analysis of each version can be found in the following subsections, while a more detailed view of the implementation can be found in the work of Canu [12].

Desktop solution

The desktop version was developed in Unity and has two main phases: the initial view and the simulation phase. The application during the first phase allows four main features, as shown in figure 2.6.

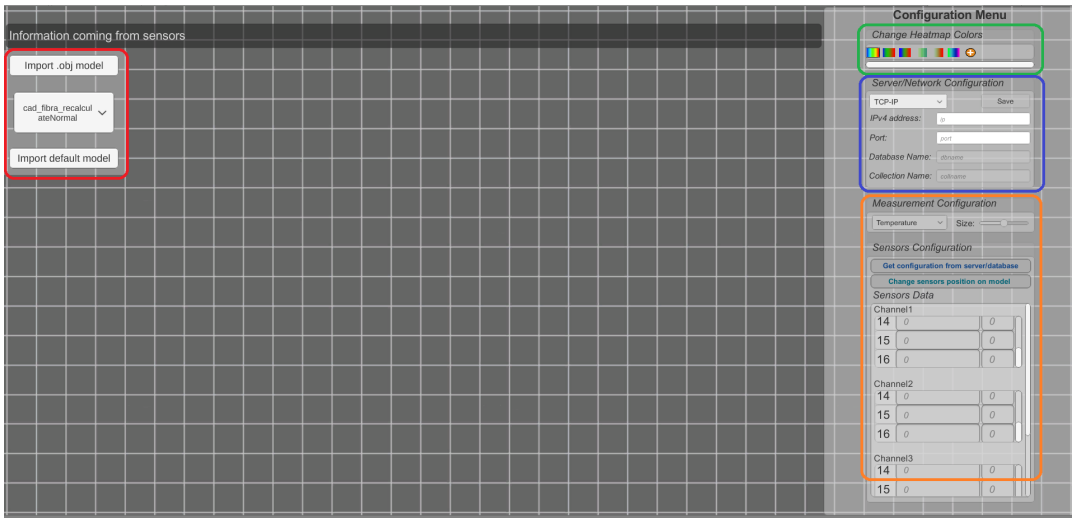


Figure 2.6: Home view of the Desktop application.

- **Import Model**, marked in red, allows to import a different CAD model rather than the default one;
- **Heat-map Customization**, marked in green, gives the possibility to select different preset of gradients for the colors of the heat-map or create a new ones;
- **Network Configuration**, with the blue bound, allows to set the connection to retrieve sensors information via a drop-down menu with three different item: TCP-IP, Real-time and Non-real-time.
- **Sensor Configuration**, the last part on the right-bottom with an orange bound, is used to retrieve sensors information from the database and change their properties, for instance the value of the idle state or the position and rotation on the model.

When all settings have been completed during the initial view, the next phase can be activated by pressing the Start Monitoring button. The simulation phase is in charge to display two different types of information:

- in the lower-left part, there is a graph with real-time information, which shows the behavior of the wavelength values over time;
- the previously customized model is displayed in the center of the screen, showing the heat-map for each sensor containing the intensity value of the data.

At the end of the simulation, the application will create some log files: one contains the wavelength values of each sensor and their latency, while the other contains a line graph as a recap of the monitoring phase.

The desktop version was analyzed to highlight possible improvements to be included in the new version developed for *HoloLens 2*. Several issues were encountered:

- one of the main observation was that, during the configuration phase, it is not possible to select both sensor types, temperature and displacement, but they must necessarily be of one type or the other. Having the ability to monitor different types of physical parameters with multiple FBG sensors has therefore proved to be a feature to be implemented;
- change the location of the sensors in the model after retrieving the data from the database it is neither easy nor intuitive. The main problem is that it is not possible to move both the model and the sensors at the same time;
- another less successful element in the configuration view is that the active sensors must be selected and added each time, through a drop-down menu and manually entering the data of which channels and gratings are currently active;
- as for the simulation phase, the graph displayed provides unclear information to the user. In fact, it only calculates the delta between the point value and the idle value. It shows all the sensors represented by the lines, which start from zero, making it impossible to recognize the various sensors, only distinguishable by the color of the corresponding line.

HoloLens solution

The AR version keeps the same structure as the desktop project, but it doesn't have all its features. In fact, only the number customization of the sensors was fully implemented. It was discarded the import of the model, the heat-map customization and it does not provide graphs during the simulation phase. The same problems analyzed in the desktop version were found in this application, with the addition of the lack of the aforementioned features. Nowadays, the application uses deprecated and poorly optimized tools, which cause drops in FPS (frames

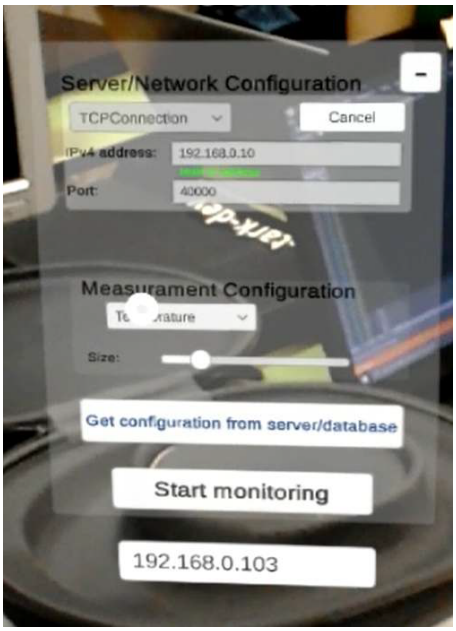


Figure 2.7: Configuration view of the *HoloLens* application [12].

per second). It was also developed with the multi-pass rendering method, which causes a lot of computational use of the CPU (more details will be explained in Section 4.2.1). As depicted in figure 2.7, the Graphical User Interface (GUI) is not very effective as, to perform the monitoring, too much data must be entered each time. For these reasons, in order to improve all the defects of the previous version, it was directly decided to start from a completely new application. The GUI was subjected to many modifications to make it easier to use and more *plug and play*. These issues makes it impossible to port on the new version of the headset created by *Microsoft©*.

Chapter 3

Photonext Viewer HoloLens Description

The following chapter describes the application, focusing on the current design and its features.

3.1 Overview

The purpose of the application is to obtain and process the flight information coming from the sensors of an airplane, so that its status can be visualized simply and immediately through the *Microsoft© HoloLens* headset. Wearing it allows the user to move and pilot the model aircraft freely while he can interact with it using hand gestures.

It is divided in three main parts:

- The Configuration phase allows to choose between creating a new sensors configuration and choosing a previously saved one, and also to select the 3D model to visualize the flight information. For the sake of simplicity, a default model of an airplane is already imported;

- The Sensors Positioning phase is triggered after selecting the *new configuration* button from the previous phase. It allows to place the sensors coming from the network connection in the actual model aircraft that can be resized and handled for a better view. The type of each sensor can be modified, switching from temperature to displacement data and vice versa. Not all sensors must be placed in the model aircraft to proceed with the Monitoring phase;

- The Monitoring phase starts after the end of the previous phase or if a saved configuration was chosen in the configuration phase. In this part, it is possible

to view the intensity of each sensor previously positioned through a heat map and a graph that shows, for each sensor, the wavelength variation.

3.2 Configuration phase

To improve the user-friendliness of the software, a ***config*** file with some information is stored in the `\UserFolders\LocalAppData\[AppName]\LocalState` folder. The application will read the contents of this file to automatically set some program operating parameters. This procedure allows the user to avoid entering the same data every time. Since the viewer can be used with different airplanes, more 3D models can be added in the `\UserFolders\LocalAppData\[AppName]\LocalState\Models` folder of the program.

3.2.1 Config file

The ***config*** file is structured as follows:

- **Connection Type**, it can be one of the following:
  - **Local** for a direct connection through the use of the TCP protocol;
  - **Database** for a direct connection to the database;
  - **Past** for streaming an old collection;
- **IP**: it must contain the IP address of the database or middleware;
- **Port**: it must contain the port of the database or middleware;
- **Username**: the username in case of database/past selection;
- **Password**: the password in case of database/past selection;
- **Database Name**: the name of the database in case of database/past selection;
- **Collection Name**: the name of the collection in case of database/past selection.

This file is standard, and if it does not exist during application startup, it is created empty at runtime. The information necessary for operation varies according to the type of communication chosen: if TCP, only the IP address and port must be completed, if instead, it is a non-real-time communication, all fields must be filled in. If the connection chosen is in real-time via the database, the name of the collection can be omitted because the program will automatically choose the most recently created collection. Unnecessary fields in the TCP and real-time phases are simply ignored.

3.2.2 Adding of a new 3D model

In case a new 3D model is added in the `\UserFolders\LocalAppData\[AppName]\LocalState\Models` folder, to work correctly it **must** have an `.obj` extension, otherwise it will be ignored.

3.2.3 Configuration Panel

The first board shown when opening the application is the configuration panel, as expressed in Figure 3.1: through it, the user can choose whether to create a new sensor configuration or to import a previously saved one (with the name of the saved configuration, name of the model used and date of creation). The selection of a saved configuration will skip the Sensor positioning phase and go directly to the Monitoring phase described in Section 3.4. The list of configurations is sorted according to the most recent save, and it can be saved only 11 types of configurations, to avoid a long list of saves and to increase user-friendliness.

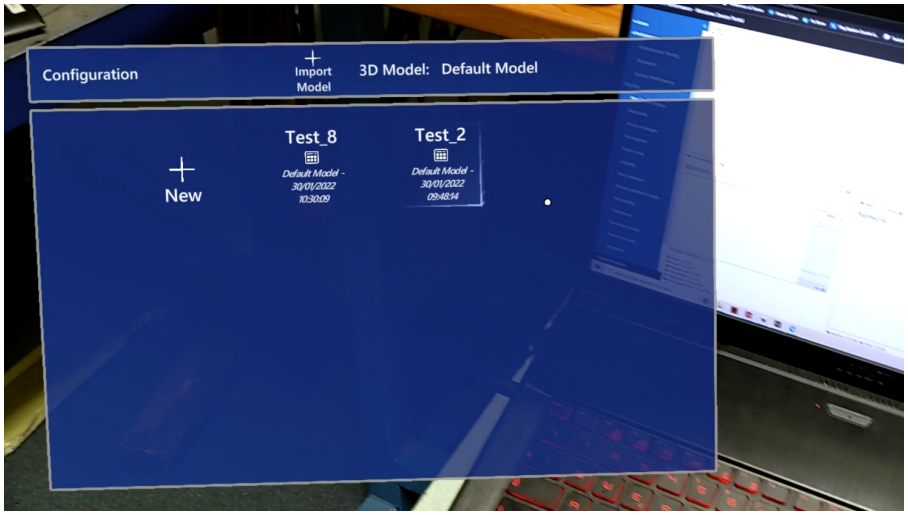


Figure 3.1: Configuration panel.

Subsequently, it is possible to import a new 3D model (Figure 3.2) which, during the Monitoring phase, will be used to display the heat map in real-time. If the import fails, the application will run with a default model. The saved configuration keeps the information of the 3D model on which the sensors have been positioned, so in case of inconsistency the user is warned.

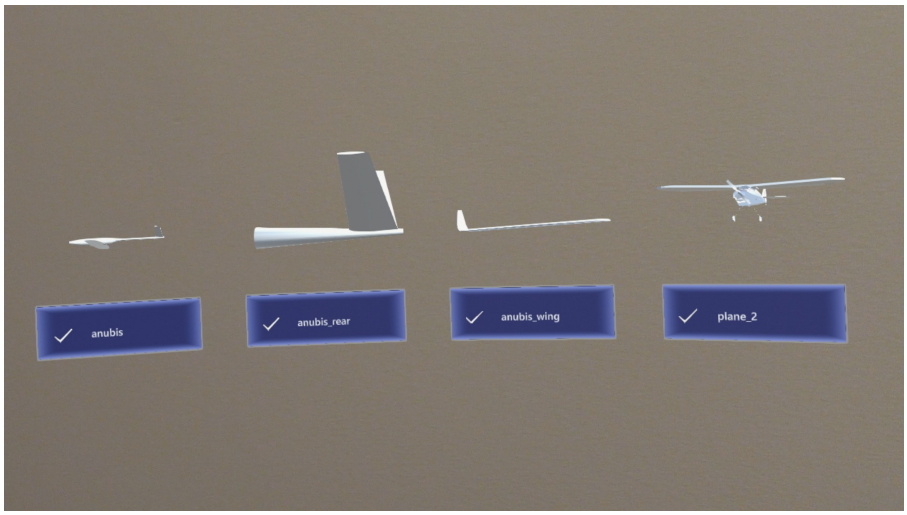


Figure 3.2: Grid with all models in the folder.

3.3 Sensors positioning phase

In this phase it is possible to choose which sensors to monitor, positioning them in the 3D model and saving the configuration obtained to use it in the future.

3.3.1 Sensors configuration

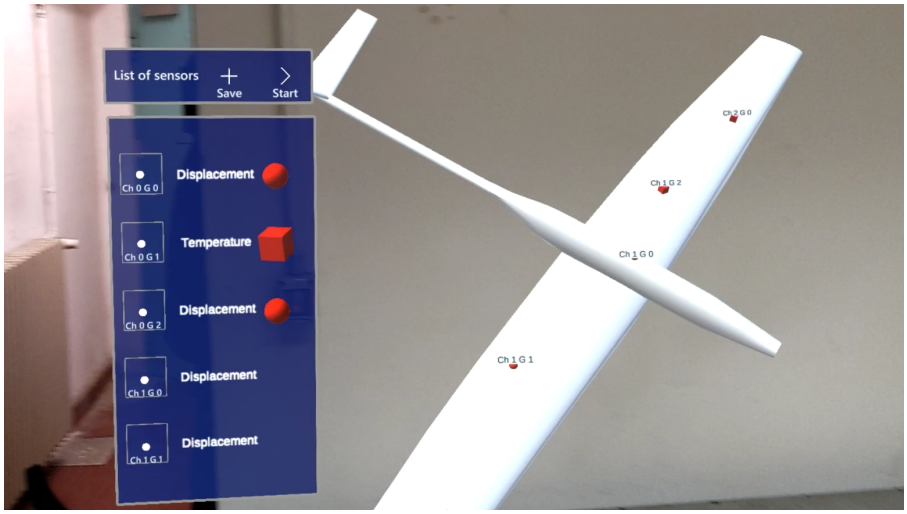


Figure 3.3: Sensor panel.

After selecting the *new* button in the configuration panel, the software will move

on to the new phase, divided into two main parts:

- On the right side it is possible to view the 3D model chosen before or, if no one has been chosen, the default one. It does not follow the user's movements and will remain statically in its position. It can be grabbed with one or two hands to move and resize it.
- On the left side there is the panel showing the sensors acquired from the database (Figure 3.3), represented by a button with a label. There is also a physical object, which can be grabbed and moved until it collides with the model, as shown in Figure 3.4.

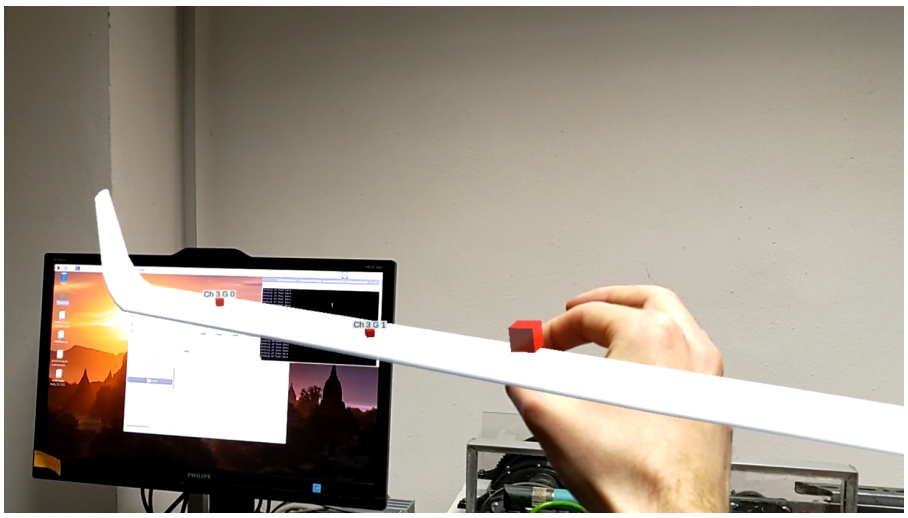


Figure 3.4: Placing sensors to the 3D model.

The panel follows the user's sight and can be grabbed by the label to move it to a position of greater liking.

For each sensor, it is possible to set the type between *Temperature* and *Displacement* by simply pressing the selection button. It is possible to intuitively understand the type of sensor based on the shape of the object: it will be a cube in the case of temperature sensors or a sphere in the case of displacement sensors.

Since the list of sensors can be quite long, only a limited number of sensors are displayed in the panel at a time. To see the others, simply scroll through the list by pointing the finger in the panel between the button and the sensor. When the object is resized and became smaller, it means that the sensor is placed in the model. However, it is possible to grab it to remove it to handle and move again; when the fingers (especially the thumb) are close to the sensor to be removed, it will become larger for an easier selection. The type button can be pressed even if the corresponding sensor has already been positioned on the model. There are also two other buttons on the panel: Save and Start.

3.3.2 Save system

Through the save button, it is possible to save the configuration obtained to use it in the future without losing time placing the sensors (Figure 3.5). This is possible even if not all sensors have been placed into the model. The configuration saved maintains all placed sensors with the correct type chosen, the model's name and orientation, and is stored with the name and the date of the creation. If the number of saved configurations is at its limit and it is necessary to save a new one, it is possible to choose an old save to be deleted: a panel is thus generated which contains a button for each saves, and when one of these is clicked, the configuration is permanently deleted, and overwritten with the new one.

The start button works whether the user chooses to save or not the configuration: in the latter case, a warning window will appear to confirm the choice.

After selecting the start button, the position and rotation of the model will remain the same during the Monitoring phase.

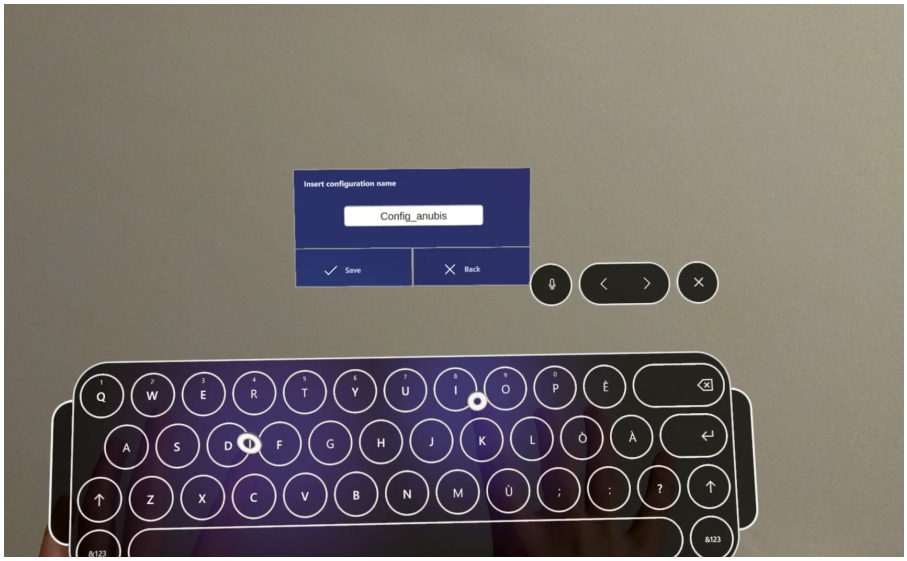


Figure 3.5: Save panel.

3.4 Monitoring phase

In this part it is possible to view the intensity of each sensor previously positioned through a heat map. It will have a different color depending on the type of sensor:

- **Temperature**: the displayed colors are, in order of intensity, green, yellow and red;
- **Displacement**: the displayed colors are, in order of intensity, light blue, dark blue and dark purple.

In the same window, a panel containing a graph is also present. It shows, for each sensor, the wavelength variation between -2 and +2 nm over time. Through this panel, it is possible to:

- hide or show both the graph and the 3D model;
- decide whether it should remain in the same position or follow the movement of the user's head.

In this panel it can also be chosen to hide or show both the graph and the 3D model, and to decide if this panel should stay in the same position or follow the user. Finally, the *Quit* button just closes the application.

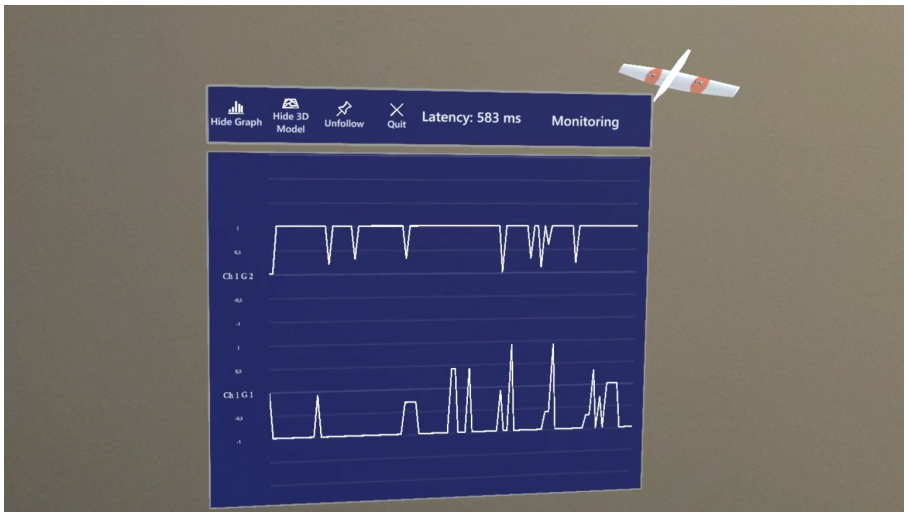


Figure 3.6: Monitoring phase: graph and the heat-mapped model on the right side.

Chapter 4

Framework and tools

During the development of the software, it was necessary to choose various existing technologies, starting from the development tools and the type of database. The idea of using mixed-reality addressed the implementation on the *Microsoft© HoloLens 2* headset. Unity engine came naturally, given the available libraries connected to the HoloLens.

4.1 Microsoft HoloLens 2

HoloLens 2 is the second iteration of Microsoft's mixed-reality device. It is a pair of smart glasses that enables a new way of interaction in people's eyes: using multiple sensors, the headset detects hands and can create holograms blended into the real world, used to display digital information. To be able to recognize the enviroment, *HoloLens 2* is designed with 4 visible light camera for the head tracking, 2 IR cameras for eye tracking, a 1-MP time-of-flight (ToF) depth sensor, a 8-MP camera and has an Inertial Measurement Unit (IMU), which features an accelerometer, a gyroscope and a magnetometer. It is build using a *Qualcomm Snapdragon 850 Compute Platform* and a memory of 4 GB LPDDR4x system DRAM. The device has also a custom processing CPU built for spatial and location processing, the Holographic Processing Unit (HPU). This allows to take the spatial mapping load from the primary CPU and offload it to the custom CPU [2].

It has a diagonal field of view of 52 degrees, an upgrade from the first version which had only 34 degree of field of view.

The *HoloLens 2* has two main gestures:

- The *Tap-air* is similar to a mouse click. It involves of a pressed stage (index up), followed by a release one (index down). With it, it is possible to grab, manipulate and resize holograms.

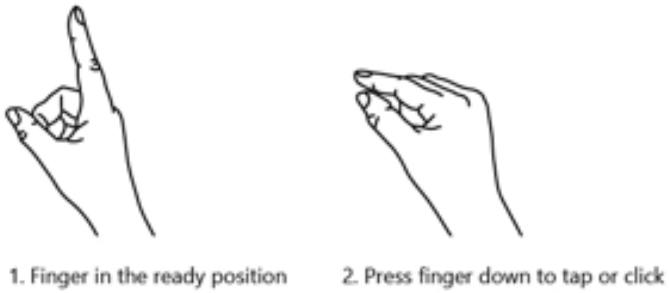


Figure 4.1: Steps to perform an air-tap gesture [13].

- The *Start gesture* opens the Start menu, useful to call the system settings, other applications, file explorer or go back and exit from an application.

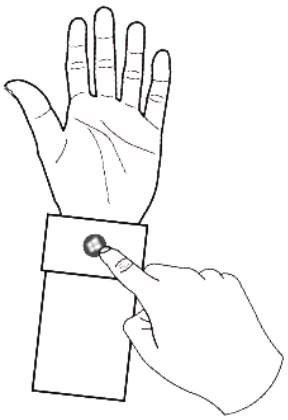


Figure 4.2: Start gesture [14].

The device is capable of understanding three types of interaction models, which suit the majority of mixed-reality experiences [15]:

- *Hands and motion controllers* allows to interact with one or two hands in a very intuitive way, removing the limits between the virtual and physical world;

- *Hands-free* allows to interact without hands, useful when the user may need to use the device while the hands are busy. The input modalities are *Voice input* and *Gaze and Dwell*;

- *Gaze and commit* is suitable when interacting with holographic content out of reach, using the gaze and commit like how people interact with computers by using a mouse to point and click. A clickable Start icon appears when the palm of one hand is facing the eyeglass view.

4.2 Unity engine

Figure 4.3: Unity logo [16]

Unity is a software framework cross-platform developed by *Unity Technologies©*, released in 2005 during the *Worldwide Developers Conference*. When it was launched, the company aimed to make the game development more accessible to developers. Now it is considered one of the most used engine specially for indie game development [17]. It is supported in more than 25 platforms, including mobile, desktop, consoles and virtual reality, becoming the ideal tool if a program has to work on different devices. Even if the software is mainly used to create three-dimensional and two-dimensional games, the engine is adopted as a production tool outside the game industry, such as film, automotive, architecture and construction for simulations and other experiences [18].

The fundamental elements of the engine are:

- The graphics rendering dedicated to the visual outcome of an interactive software. Unity implements with its own real-time rendering pipeline the global illumination, ray-tracing and physical based rendering;
- The physics system used to handle physical simulation in real-time. The one used by Unity is *PhysX©*, developed by *NVIDIA©*;
- The scripting is an essential part of the application. Scripts are used to respond to inputs from player and to create relationship between the elements in the scene;
- The Graphical User Interface (GUI) and a set of tools provided to build one.

The main part of the development in Unity is scripting using C# language, but all the entities in the scene (3D objects, camera, light, etc.) are represented with a base class: a `GameObject`. It is used as containers of *components*, a base class for everything attached to the `GameObject`. Each component describes how

a `GameObject` should behave, for example, the default component `Transform`, always attached to every `GameObject`, set the position, rotation and scale of the `GameObject` itself. To store coordinates position, an important classes should be introduced: `Vector`. Vectors are often used to describe some of the fundamental properties, such as the position of a character, the speed at which something is moving or the distance between two objects; it can be expressed in multiple dimensions, and Unity provides the `Vector2`, `Vector3` and `Vector4` classes for working with 2D, 3D and 4D vectors. These three types of Vector classes all share many of the same functions.

The most commonly used function in scripting are `Awake` (the very first function called before the application starts), `Start` (similar to the previous one, but called each time the script is enabled) and `Update` (called at each frame update) [19].

In order to create a robust software it was necessary to use external tools, compatible with Unity, well known and already tested by Unity developers:

- XR SDK is a plug-in framework that enables XR providers such as Microsoft HoloLens to integrate with the Unity engine and make full use of its features;
- Mixed Reality Toolkit is a software development kit, was developed as a consequence of the release of the *HoloLens 2*. The SDK gives all the useful tools to develop mixed-reality applications with Unity in a simple way;
- Obj Importer is a simple plug-in that allows to import `.obj` models with a given path;
- Graph and Chart is a plug-in developed for 2D/3D data visualization.

The Unity version used for development was 2020.3.19f1 (Long-Term Support).

4.2.1 XR SDK

Since the Unity version 2019.3, the company unified plug-in framework to have direct integration for multiple platforms, and have common features to support XR hardware. In addition to other various platforms, it supports *Microsoft© HoloLens 2*, with the *Windows Mixed reality* [20].

One of the features of XR is the *Stereo Rendering* mode for VR and AR headset. For Windows Holographic devices such as HoloLens [22] there are two types of rendering methods:

- **Multi-pass** rendering, which performs two complete render passes, one for each eye. The problem of this mode is that it generates the double CPU usage compared to the following method, but it don't require any code changes for the developer;

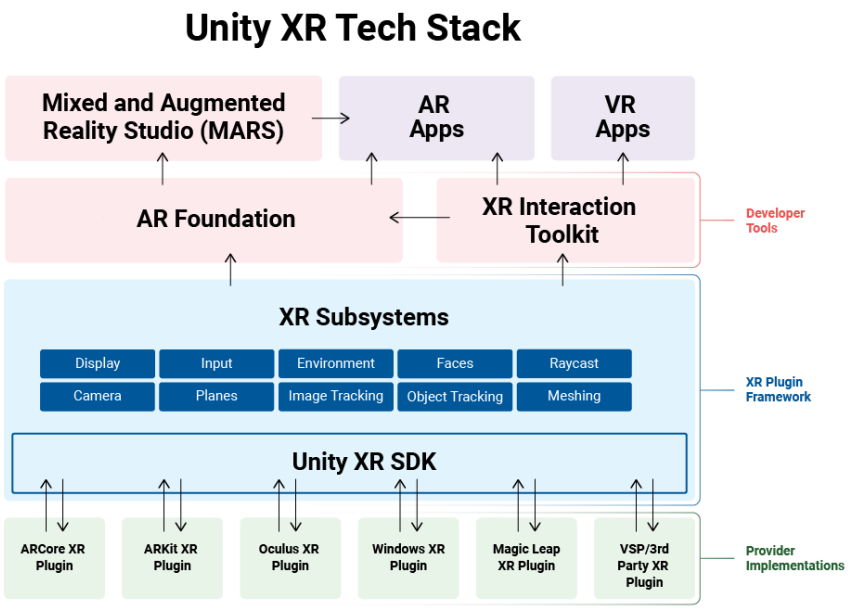


Figure 4.4: Unity XR plug-in framework structure [21].

- **Single-pass instanced** rendering, which heavily decreases CPU workload because it only performs a single render pass where each draw call is replaced with an instanced draw call. During this rendering mode, both eyes share the work required by culling and shadow computation. Unity's built-in rendering features support this feature, however custom shaders have to be adapted to run with this rendering mode.

The XR settings used for development was:

- Mixed Reality Depth Format - 16-bit depth;
- Windows Mixed Reality Depth Sharing - enabled;
- Initialize XR on startup - enabled;
- Stereo Rendering Mode - single-Pass Instanced.

4.2.2 Mixed Reality Toolkit

MRTK-Unity is a Microsoft-driven project created for building mixed-reality experience for Virtual Reality (VR) and Augmented Reality (AR). The toolkit is used to accelerate cross-platform app development with a unified set of components and

Figure 4.5: MRTK logo [23].

features. The usage of the toolkit was essential to support development thanks to the provided packages for user experience.

Here some remarkable examples of building blocks for spatial interaction:

- **Button** is a basic component for interaction. When pressed, it triggers an immediate action. Since icons and text are difficult to read in a physical environment, the buttons have an opaque back plate, to avoid usability and stability issues. There is also a front plate in the button not visible during idle state, but when a finger reaches the surface, the front plate's bright edge becomes visible. When pressed with a finger, the front plate shifts with the fingertip. There is also a quite pulse reaction when the fingertip touches the button to give a visual feedback of the contact point.

Figure 4.6: Buttons with front plate highlighted [24].

- **Object manipulator** is a component that makes a `GameObject` manipulable with the hands. It can be movable, scalable and rotable. Along with this script, a `NearInteractionGrabbable` component is added. This makes the object grabbable with a near interaction with hands. The object needs a `Collider` element that matches the grabbable bounds. The type of manipulation can be with one, two or both hands and the object can rotate about its center or the grab point. The script has a lot of properties, including

the smoothing of the movement, the constraint of the manipulation and the possibility to add physics to the behavior of the object.

- **Keyboard** can be invoked by the application at any time: this tools provides a hologram of a keyboard compatible with the virtual environment, that can be used with multiple fingers.
- **Dialog** prefab is a UI panel that asks for the user's confirmation or provides acknowledgment. It is used to notify important information before an action can be completed. It can spawn at a certain distance for far or near interaction, and it can be customized with some type of confirmation buttons.
- **Object collection** is a script that helps visualize an array of objects in three-dimensional shapes, such as plane, sphere, cylinder and radial. It automatic aligns elements to the designed shape each time a new one is added to the group. This component can be integrated with a `ScrollingObjectCollection`, which enables scrolling of three-dimensional object through a contained viewable area.

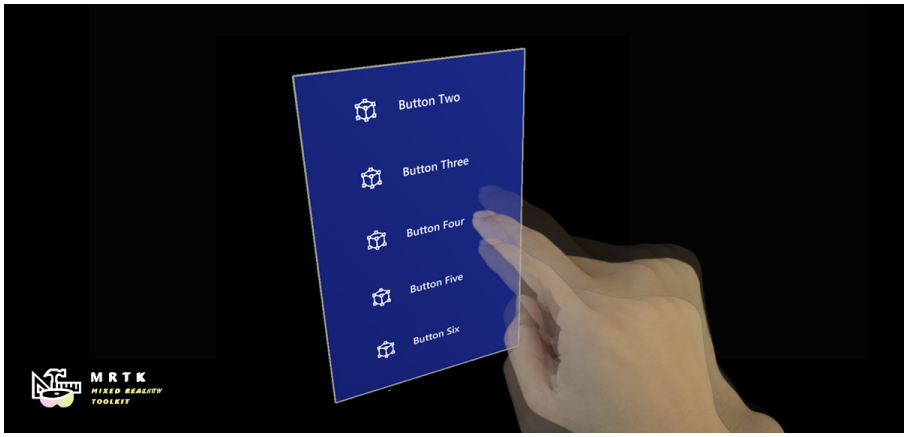


Figure 4.7: Scrolling collection object [25].

- **Solver** is a component that makes easier to calculate the position and the orientation of objects according to a predefined algorithm. There are some types of solvers that provides different basic behavior:
  - `Orbital`: locks to a specified position and offset from the referenced object;
  - `ConstantViewSize`: scales to maintain a constant size relative to the view of the referenced object;
  - `RadialView`: keeps the object within a view cone cast by the referenced object;

– `Follow`: keeps the object within a set of user defined bounds of the referenced object.

More information about the features can be found in its documentation [26] and in source code [27].

4.2.3 Obj importer

One of the features already existing in the desktop solution but not present in the previous *HoloLens* version was the possibility to import a 3D model after the project's build. This is because the end user does not have to learn Unity to update the application, so that can work with a different model. The application is stable and packaged: there is only a button that allows choosing a new model previously inserted in a predefined directory. This action is easier to perform than modifying the asset within the project, rebuilding the application and moving it to the device. To realize this feature was necessary to use an external script that automatically imports a `.obj` file as a `GameObject` [28]. When the user clicks on the *Import model* button, a radial view with all models located in the default folder appears and the user can select the new element to import. For each model in the given folder path, the `Load` function of the `OBJLoader` class is called, which extracts all the fundamental data such as vertices, triangles and other elements that need to represent a 3D model in the scene.

4.2.4 Graph and chart

Graph and chart is an external plug-in for Unity useful for real-time updated graph [29]. During the monitoring phase, it is important not only to have eyes on the heat-mapped 3D model, but also to have an overview of the wavelength variation for each sensor during time to correlate the last with past measures. The graph is the key to display how sensors data evolve in time: each line in the graph describes the change in the wavelength (y-axis) of a given sensor over time (x-axis). The main reason to use this asset is that it is the best tools in Unity for large amount of real-time data, while develop and implement a simpler version from the beginning may cause a some issue. So, instead of including a new functionality, it was adopted the best tool already fully tested by a large number of developers. More information about the tool can be found in its documentation [30].

4.3 MongoDB

MongoDB® is a document-oriented database, a computer program and data storage system designed for storing, retrieving and managing information. It is classified as a NoSQL database, which differs from relational database as it provides a different

Figure 4.8: MongoDB logo [31]

mechanism for the storage and retrieval of data, avoid using the tabular format. A JSON-like documents format with optional schemes is used to store data. It was developed by *MongoDB Inc.* and licensed under the *Server Side Public License*.

One of the reasons why it is adopted is due to its capability of storing data using flexible models, which are more efficient than a traditional database like *MySQL* [32]. It supports a wide variety of use cases across industries and it can scale to handle large volume of data and a big number of users, thanks to the support for embedded data models and indexes. Compared to other NoSQL database, it shows better results for retrieving multiple documents and inserting documents with multiple threads [33]. MongoDB also provides a large query language, supporting not only the typical instruction such as the insert, select, update and delete, but also aggregations and text search. Thanks to the *Sharded Clusters* components, it can have horizontal scalability and, due to the replica set, it can also have a high availability, since provides data redundancy and automatic fail over.

Chapter 5

Implementation details

In order to understand the previously proposed and described ideas, it is worth discussing the aims and methods of the implementation.

5.1 Class structure - Class Overview

The application is divided in different classes. The core of the project is structured in two macro classes: `GameManager` and `GUIManager`. The first one is in charge to manage all the others and to handle the transition between phases, from start up to configuration and finally monitoring phase. `GUIManager`, as the name suggests, is used to direct all the parts related to the graphical interface, including the graphic and interactive panels, the heat-mapped model and the graphs.

Then, there are two important classes that deal with the management of the network communication:

- `MongoDBManager`
- `TCPManager`

`UnityMainThreadDispatcher` is one single instance of a class with the thread management role. Other scripts have been created to manage how objects interact with each other or for specific purposes, such as managing the behavior of the 3D model or the sensor manipulation procedure. At last, the class `Sensor` is a struct that contains all the information of each FBG sensor:

- `Channel`;
- `Gratings`;
- `MeasurementType`;
- `Wavelength` (idle state);

- `Wavelength` (value to update);
- `Position` (coordinates for the location of the sensor when placed in the 3D model).

All sensors are stored in a `Dictionary` variable that has the index of each sensor as its key, and the associated `Sensor` class as its value.

The class structure was designed trying to maintain continuity with respect to previous versions of the desktop application, which was a point of reference for the new project. In the next subsections the most important classes will be examined, while in the following sections the methods of implementing particular and noteworthy features will be focused on.

5.1.1 GameManager

It is the main class and the one that directs, because it keeps all the relevant information that all other scripts can always refer to. It is designed with a *singleton* pattern, which means that this class is limited to a "single" instance, and it is the first custom script which is executed to ensure correct initialization of the other components. The variables created by `GameManager` are static, which means that are all unique variables that cannot have different instances or values. However, this class is not only the container of static variables but it also contains the methods to manage the other scripts. In particular, it manages the operation flow and the life of the various phases: initialization, start, sensor configuration management and start of the monitoring phase. The functions that guarantee the passage between phases are `Start`, `StartUp` and `StartMonitoring`. The functions associated to the setting of the variables are:

- `SetSensorsConfiguration`, that is a method to set in the dictionary variable the sensors information;
- `SetMonitoringSensors`, that is a method of updating the dictionary with only the sensors selected during the configuration phase;
- `UpdateSensorIdle`, that is a method to update the value of the wavelenght during idle state;
- `UpdateData`, that is a method to update the heat-map and the graph with new values from the middleware.

5.1.2 GUIManager

The most important class after `GameManager` is the one that deals with the graphical interface: `GUIManager`. Based on user's choice, it is in charge of showing, activating

or deactivating parts of the interface. It is divided into several sections, which correspond to various internal phases of the behavior of each element.

During the configuration phase, it generates the panel with the list of all the saved configurations. It also handles the event click of each button in order to load a previous configuration and manages the choice of creating a new configuration, which leads to the sensors positioning phase. In this panel, there is also the button that allows to import a new model. When clicked, the script deactivates this panel and generate buttons and 3D models associated with all the files contained in the default folder.

In the sensor positioning phase, `GUIManager` creates the panel with all the sensors retrieved from the database and shows the previously chosen 3D model at a certain size, also allowing its manipulation. In this phase, it is also responsible for managing the saving of the configuration, thus showing a keyboard to give a name to the configuration just created.

During the monitoring phase, it shows the component associated with the graph by recurring to the `ActivateGUIMonitoring` function, handles the updates of the values for both the heat-map and the graph, and allows the user to show or hide both the graph and the model. In addition to the material management of these panels, the class also manages all the dialog windows that are created to warn the user throughout the life of the software.

5.1.3 UnityMainThreadDispatcher

The `UnityMainThreadDispatcher` class, like the `GameManager` class, implements a *singleton* pattern, in which for the entire life of the application there is only one instance of the same script. It contains a list of functions required by the secondary thread to be executed by the main one: whenever the main thread calls a new request, it is placed in a queue as an `Action` object, which guarantees the order of service as the order of arrival (*FIFO*). In fact, in the `Update` method, the oldest queued function is removed from the queue when needed. The script was created by De Witte: further details of source code can be found on [34].

5.1.4 MongoDBManager

The `MongoDBManager` class, as the name suggests, is in charge of managing the connection to the database and selecting the desired collection. After this passage, it retrieves the list of sensors marked as active from the database: these are used to be listed in the panel during the positioning phase of the sensors. In the most recent version of the middleware there is no longer the distinction between active and inactive sensors, but all those present in the database are active by default. The script also contains two types of functions for reading in real-time from the

database or for reading an already existing collection in non real-time. Further details of the functions are explained in Section 5.3.

5.1.5 TCPManager

The `TCPManager` class is similar to the previous one: it handles network communication with the middleware client, but through a direct TCP connection instead of using a database connection. The connection is via LAN, so both systems must be connected to the same network. There are two types of TCP listeners:

- one is in charge of serving the configuration packets, which are the sensor information from the middleware;
- the other is in charge of serving the data packets, which are the wavelength value read from the interrogator.

More details are described in Section 5.4.

5.1.6 FileConfig

The `FileConfig` script reads the network communication configuration settings from a file. To increase user-friendliness, it is suggested to edit this file through a PC and to move it to the HoloLens before opening the application for the first time, to avoid managing many settings from the headset. The `ReadConfig` function reads the first line of the file to understand what type of connection is required:

- real-time with the database connection;
- real-time with TCP connection;
- non-real-time only with the database connection.

After that, it reads the remaining lines and inserts the information into the appropriate struct. Depending on the type of connection some lines are ignored and, in case that connection is with the database, its name is not needed as the program looks automatically for the most recent collection. The file data are first formatted to be readable in a standard way, and then a first simple check of the validity of each string is performed: in the case of an IP address check, the coherence of the address is checked by regex with the `CheckIpAddress` function, while in the case of strings of names the function `CheckStandardString` checks for the possible presence of special characters not allowed. The `ReadUserData` function manages the reading of rows in an unsorted way, so that two data are considered valid even if they are inverted with respect to the standard position in the file.

5.1.7 SensorManipulation

This script handles the sensor manipulation: it has four main functions:

- `OnHovered`;
- `OnHoveredExited`;
- `StartManipulation`;
- `EndManipulation`.

The first two respectively realize the feature to enlarge and reduce sensors when the user's hand is near the object. On the other hand, the others handle the event called when the user grabs and release the sensors. The algorithm for positioning the sensors in the model is explained in detail in Section 5.6.

5.1.8 SensorObject

`SensorObject` is a component associated to each sensor and handles the user event click of the sensor panel to change their type. The function called, `OnChangeSensorType`, inverts the value of the parameter `measurementType`, contained in the dictionary of the selected sensor, then deactivates the corresponding 3D object and activate the correct one. The sensors dictionary simply collects information of all the sensors during the run.

5.1.9 MonitoringModel

This script is a component associated only with the `Model GameObject`. Every time a new model is imported and selected as the reference model, it adds the components necessary for its working, such as `ObjectManipulator` and `Rigidbody`. Independently from the measure of the `.obj` models, it also resizes them in order that they appear always proportionate.

5.2 Multi-thread communication

As already introduced, the program works in a multi-thread environment. Its main thread takes care of executing the *Unity Pipeline*, handling the GUI and the object behavior. Instead, the secondary ones manage the network communication. Depending on the choice of connection type, the generated threads are different. In the case of real-time choice with the database, only one secondary thread is created: this takes care of calling the dispatcher in order to subsequently update the sensor list.

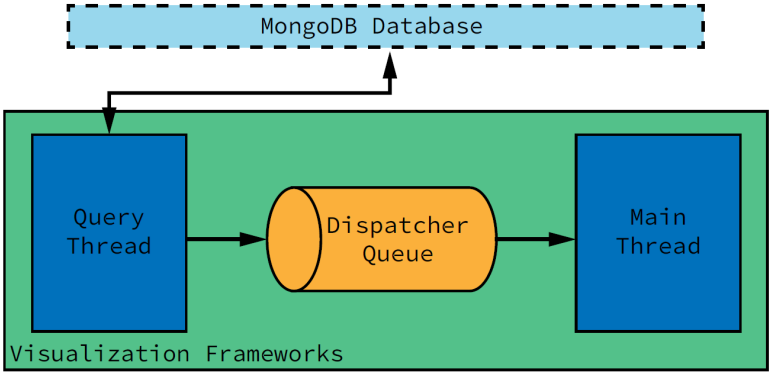


Figure 5.1: Multi-thread connection schema for MongoDB [12].

When the connection with the TCP is chosen, two different secondary threads are instantiated. These correspond to the handling of the `ConfigPacket` messages and the `DataPacket` messages. Thanks to this management, the parallelism and efficiency of the application are improved.

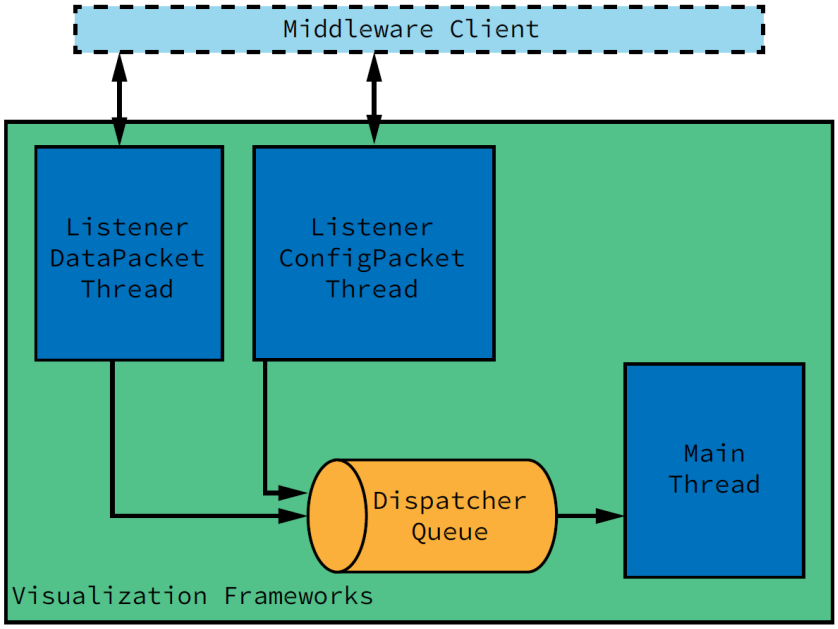


Figure 5.2: Multi-thread connection schema for TCP Connection [12].

5.3 Database communication

As previously mentioned, the communication via database is performed with MongoDB. Thanks to the database notify mechanism, it is possible to subscribe to the desired collection and obtain the data in real-time. *Microsoft©* releases a MongoDB C# Drivers as DLL. These can be easily added to the Unity project as plug-ins by just copying them into the project directory. Sadly, these drivers seems does not work with Unity *IL2CPP* builds. Custom build of the drivers were made thanks to the work of Bock [35], so it was possible to import them to make the project work. Storing the data in the database also allows to review them at another time, not in real-time. The `GameManager` script, after reading the config file and verifying that it was possible to connect to the database, calls two functions that work in both real and non real-time cases. `InitMongoDB` takes care of setting the client with username, password, IP address and port, and performs a ping to determine if the connection has been established. `GetSensorConfiguration` queries the database and retrieves all active sensors. The data written in the database are divided between *config* type and *peakData* type: the former are used to make a quick query to understand which sensors are going to be analyzed, while the latter are the real-time readings coming from the middleware and consequently from the interrogator. Figure 5.3 shows portion of `GetSensorConfiguration`: the query is filtered by type and activity, then the data are sorted. Next, iterating over the result, the sensors are added to the dictionary variable in GameManager.

```
var filter = Builders<BsonDocument>.Filter.Eq("type",
   "config") & Builders<BsonDocument>.Filter.Eq("is_active",
   true);
var sort = Builders<BsonDocument>.Sort.Ascending("channel")
            .Ascending("grating");
var results = collection.Find(filter).Sort(sort).ToList();
foreach (BsonDocument doc in results) {...}
UnityMainThreadDispatcher.Instance().Enqueue(new Action(() =>
{
    GameManager.SetSensorsConfiguration(sensors);
}));
```

Figure 5.3: Query to retrieve all active sensors.

5.3.1 Real-time data

The real-time section is defined with a single function, called `StartWatch`. To obtain the data in real-time via the database, the Change Stream feature provided

by MongoDB was used: thanks to this, it is possible to subscribe to a collection and be notified every time the tables are updated. To subscribe to the feed, the program needs to open a *changestream pipeline* in which a *cursor* reacts to changes. Figure 5.4 shows how the options and the pipeline are structured.

```
var options = new ChangeStreamOptions { FullDocument =
    ChangeStreamFullDocumentOption.UpdateLookup };
var pipeline = new
    EmptyPipelineDefinition<ChangeStreamDocument<BsonDocument>>()
     .Match("{operationType:{$in:['insert']}}");
using (var cursor = collection.Watch(pipeline, options)) {
     ...
     var result = cursor.Current;
     foreach (var elem in result)
     {
          //Add sensors information and update
     }
}
```

Figure 5.4: ChangeStream options.

After opening the *changestream* and receiving an update notification, the program must iterate with the *cursor* to find all the new updates of the collection. In the first iteration, it is necessary to save the received wavelength values as the idle state: this becomes the reference for for calculating the delta of the next values. In the following iterations, the wavelength values for each sensor are saved, and the intensity is calculated with respect to the idle value. The sensors information is saved in a variable of type `Dictionary<int,Sensor>`, while the properties for updating the heat-map values are stored in an `Array`. When it is time to update the visual information (graph and heat-map), decided before building the application (it was decided to set one second to this value), the average latency of the data is calculated, and then the `UpdateData` function is called, which is in charge of updating the information.

5.3.2 Non real-time data

The non real-time mode is quite different from the real-time implementation. It only keeps the final structure similar, more precisely the part in which the information is saved and the `UpdateData` function is called. In fact, to manage the replay of a collection, the first thing to memorize is the first idle values stored in the database. The `FirstBatchData` function called by the `GameManager` script is in charge of retrieving this information. Next, the `GUIManager` script synchronize

```
radius = GameManager.modelScale * GameManager.heatMapScale;
intensity = (Math.Abs(wavelength -
   sensorInfo.WavelengthIdle)/GameManager.globalMaxVariation);
sensorInfo.Wavelength = wavelength;
_sensors.Add(id, sensorInfo);
properties[id] = new Vector2(radius, intensity);
if ((now - timePassed) > GameManager.millisToUpdateGraph)
{
    ...
    UnityMainThreadDispatcher.Instance().Enqueue(new
       Action(() =>
    {
        GameManager.UpdateData(_sensors,
           properties.ToArray(),latencyTime);
    }));
}
```

Figure 5.5: Storing data and update.

```
public bool GetLastValues(long t, Vector4[] properties)
{
    if (results.Count == 0 || results.FindIndex(x =>
       x.GetValue("curr_time").AsInt64 >= t) >=
       (databaseEntries * 0.8f)) GetPastData(t);
    var DistinctSensors = results.Where(x =>
       x.GetValue("curr_time").AsInt64 <= t).GroupBy(y =>
       y.GetValue("index").AsInt32).Select(z => z.Last());
    foreach (BsonDocument doc in DistinctSensors)
    {
        //Add sensors information and update
    }
}
```

Figure 5.6: `GetlastValue` function, extracts more recent values respect to the given time.

itself by counting the elapsed time, so that it will only call the data request function (`GetLastVaues`) after a certain period. The `GetLastValues` function is responsible for finding the most recent data compared to the time synchronized by the `GUIManager`: 10000 data are stored in a `List<Bsondocument>` variable, but only the most recent ones must be used at the time of the call.

Since there can be many documents in the database, it was decided to keep a few entries of the database in memory, on one hand not to have the entire database in memory, but also to avoid having to query the database too often, so as not to overload the connection and the database. For this reason, the `GetPastData` function is in charge of making queries to the database only after a certain period of time.

```
public void GetPastData(long t)
{
    var filter = Builders<BsonDocument>.Filter.Eq("type",
       "peakData") &
       Builders<BsonDocument>.Filter.Gte("curr_time", t);
    var sort =
       Builders<BsonDocument>.Sort.Ascending("curr_time");
    results = collection.Find(filter).Sort(sort)
                .Limit(databaseEntries).ToList();
    ...
}
```

Figure 5.7: `GetPastData` function, gets a portion of the database filtered by time.

5.4 TCP communication

Connection using MongoDB is very useful for saving data in the database and analyzing it later, but it does not guarantee acceptable latency. In fact, the time when the data are read and processed by the middleware, sent to the database and then read by the application is near real-time and does not allow immediate analysis. Consequently, it was decided to implement direct communication in the same local network also in the new version, through TCP packets. This modality allows to reduce latency, but its actual implementation is quite challenging. One of the possible ways is to mount an omnidirectional antenna over the model aircraft and a directional antenna on the ground. This allows create a Wi-Fi connection for sending data in a local network.

The script has two different regions, one reserved for the server and the other for the client. These communicate with each other by requesting packets or replying with the requested data. `InitTCPListener` initializes the communication method by generating two thread listeners. One menages the part relating to the configuration (analogous to `GetConfiguration` for the connection to MongoDB) and the other handles receiving of the `peakData`.

The protocol is designed to provide six types of packets, two for sending data and three for requesting data:

- `RequestConfig` is a message send to retrieve to the middleware client configuration packet;
- `ConfigPacket` carries the list of all active FBG sensors;
- `RequestDataStart` requests to start the monitoring phase;
- `Data Packet` carries the data from last frame;
- `RequestDataEnd` requests to stop the monitoring phase;
- `EndThreadPacket` is sent to stop the execution of the TCP server.

The content of the datagram is divided into a header part that contains the packet id and eventually the payload. This is used only in the case of ConfigPacket and DataPacket packets.

The DataPacket payload is 768 bytes long and contains a list of 64 sensors, in which each element encode two variables: the sensor sample as a 32-bit float and the timestamp in microseconds as a 64-bit integer. If the sensor is not active, both values are left to 0.

The ConfigPacket payload is 1600 bytes long and is divided into 64 units of 25 bytes in which the configuration of each sensor is stored. The content of the 25 bytes is described in detail in Table 5.1.

Bytes range	**Description**
0-4	Index of the FBG sensor, which is a 32-bit integer
5	Boolean value to specify the activity of the FBG sensor
6-8	Wavelength value in idle state, encoded in 64-bit float
9-12	Wavelength variation, encoded as a 64-bit float
13-24	Coordinates for sensor's location (x,y,z each one ias 32-bit float)

Table 5.1: ConfigPacket payload.

The request packets are sent through the Send function, which is responsible for connecting to the server and writing the requested data type to the buffer. The two listeners are always active in two different threads and start two different methods as soon as they receive an appropriate request from the client:

- `ServeConfigPacket` reads from the stream the packet containing all the data of the active sensors and fills the values received in the `Dictionary` variable `sensors`;

- `ServeClientData` reads the wavelength values for each sensor from the stream until there is a stop request and saves the values in the same format of the database.

5.5 Configuration phase

This section examines the most important methods and algorithms of the initial configuration phase, in particular the mechanisms for saving a new configuration and for importing new 3D models.

5.5.1 Save and load configuration

The `save.txt` file contains all the information regarding the chosen configuration. It is composed of all the information necessary to recall the chosen configuration. To do this, there is a class called `SaveSystem` with the `Save` and `Load` functions, which respectively write and read a string in *JSON* format containing all the data. The *JSON* format is a text-based way of representing objects. It has become popular because it is human-readable, compact and lightweight. To be able to store a configuration it is necessary to save several main values, such as name, creation date, associated 3D model and its rotation. Furthermore, it is necessary to memorize the list of all the sensors placed alongside their type and coordinates. The `Configuration` class keeps track of this information: it contains the general information of a save mentioned before and a list of sensors with their own properties, represented as a `List<Model>`. The `Model` class contains only the information necessary to save a sensor, so it is a lightweight version of `Sensor` class. All the `Configurations` are stored in a `List<Configuration>`. Figure 5.8 shows the structure of the classes in detail.

All configurations, before being saved, are converted into *JSON* text using the `ConvertConfigurationsToJson` function: it sorts all configurations according to the most recent creation date and copies the list into a `SaveObject` object, used for conversion. Thanks to the `JsonUtility` class, the `SaveObject` object is converted into a string.

The information to create a save is collected by the `SaveConfiguration` function. This function loops for all the children of the `GameObject Model`: each child is a sensor positioned in the model, so iterating through it saves its local index, coordinates, rotation and scale. Another important data to save are the coordinates of the contact point between the sensor and the plane surface, which will be provided to the heat-map. Saving these coordinated avoid to calculate them each time and improves the program performance.

At the end of the save function, the button that allows saving in the sensors panel became unusable, but as soon as a sensor is added or removed or the position

```
public class SaveObject
{
    public List<Configuration> configurations = new
       List<Configuration>();
}
public class Configuration
{
    public string name;
    public string date;
    public string modelName;
    public Quaternion modelRotation;
    public List<Model> sensors = new List<Model>();
}
public class Model
{
    public int index;
    public int sensorType;
    public Vector3 pos;
    public Quaternion rotation;
    public Vector3 scale;
    public Vector3 heatmapPos;
}
```

Figure 5.8: `SaveObject` class, `Configuration` class and `Model` class.

of an already present one changes, it becomes possible to click again and save the different configuration again. In case all save slots are occupied, an old one must be deleted to allow the creation of a new save. The `DeleteOldSave` function handles this feature by generating a panel similar to the save one but in this, the click is associated with the deleting of the same selected. After the configuration is removed, the *JSON* file is updated and the `save.txt` file is overridden.

The `LoadConfiguration` function is in charge of converting the contents of the `save.txt` file from the *JSON* format to the `SaveObject` object. It will be processed by the `GenerateModelSavedGrid` function (contained in `GUIManager`) that handles the generation of the panel with the list of saves.

5.5.2 Import model

In the first HoloLens version there was no possibility to select a different model than the default one. For this reason it was decided to add this feature to allow user to perform monitoring activity also with other model aircrafts, trying to maintain a

good level of ease of use and immediacy, minimizing the interactions that the user must do. The models are added inside a default folder before running the program, and they are shown if the user decides to use a different model. The `OnClickImport` function hides unnecessary interface elements and generates a radial grid around the user (using the component provided by *MRTK*, `GridObjectCollection`). Each element of the grid consists of a button and one of the 3D models available in the folder. The files are read using the `GetModelsFromDirectory` function: this method loops through all the `.obj` files, which are loaded into a `GameObject` list with the Load function thanks to the `OBJLoader` plug-in previously described in Section 4.2.3.

`NormalizeModel` function allows to show all the models by using similar proportions. It calculates the largest side of the model, and then all the sides are divided by this value, making the model unitary in size. Subsequently, the model is scaled to a standard size, given by the input parameter of the function itself.

```
Bounds bounds = new Bounds(model.transform.position,
   Vector3.zero);
foreach (Renderer renderer in
   model.GetComponentsInChildren<Renderer>())
{
    bounds.Encapsulate(renderer.bounds);
}
var size = bounds.size;
float unit = Mathf.Max(Mathf.Max(size.x, size.y), size.z);
model.transform.localScale /= unit;
model.transform.localScale *= scale;
```

Figure 5.9: Algorithm to resize models to the same width.

When the user clicks on one of the buttons present in the import menu, the `ImportModelName` function destroys the old model and sets the new one. It also inserts the components necessary for its operation, such as `ObjectManipulator` and `RigidBody`. Subsequently, the import menu is hidden and the program returns to the configuration panel.

5.6 Sensor manipulation

The manipulation of the sensors allows the user to move them from their panel to the reference model, positioning them in a way which should be similar to the physical one, so that the associated heat-map can be viewed during the monitoring

phase. When they are released from the hands, sensors are binded automatically in the closest point of the model.

Two functions handle this feature:

- `EndManipulation`;
- `SetSensor`.

`EndManipulation` is called when the user releases the object from the grab and is in charge of calculating the coordinates of the closest point of the model with respect to the position of the sensor. The `ClosestPoint` function is an internal method of the `Collider` component. To work correctly, `Collider` must be set to convex and its type must be one of these types: `BoxCollider`, `SphereCollider`, `CapsuleCollider` or `MeshCollider` but with the *convex* option selected. The 3D model can be structured by several `MeshColliders`. For this reason, the function iterates for each existing `MeshCollider` to find the closest point among all the closest points (with the `ClosestPoint` method) related to the i-th `MeshCollider`.

```
MeshCollider[] meshColliders =
   ModelStructure.GetComponentsInChildren<MeshCollider>();
for (int i = 0; i < meshColliders.Length; i++)
{
    Vector3 tempPosition =
       meshColliders[i].ClosestPoint(transform.position);
    float distance = Vector3.Distance(transform.position,
       tempPosition);
    if (distance < Maxdistance)
    {
        Maxdistance = distance;
        position = tempPosition;
    }
}
Vector3 startingPosition = transform.position;
Vector3 direction = (position - startingPosition).normalized;
```

Figure 5.10: Algorithm to find closest point and its direction.

The `SetSensor` function is called after finding the coordinates of the nearest point and manages the connection of the object to the surface. Since the Closest-Point method exploits the convex MeshCollider, coordinates are not accurate but performance are higher. For this reason, the calculated coordinates only serve as a direction in which to project a `Raycast`.

`Raycast` is a virtual ray that creates a line from an origin with a given direction. It is able to detect any collisions, so it is used to detect if an object is present in its direction. If the ray collides with the model, then it is possible to translate the position of the sensor in the coordinates of the collision point, adding an offset due to the fact that the *pivot* of the sensor is the center of the object, while the contact point is on its surface. The object becomes a child of the `GameObject Model`, so that it remains in the same local position with respect to the parent one.

```
if (Physics.Raycast(startingPosition, direction, out hit))
{
    //Hit case
    transform.SetParent(ModelSensor.transform, true);
    transform.localScale = transform.localScale /
       scaledObject;
    transform.position = hit.point -
        (-hit.normal *
           transform.GetChild((int)GameManager.sensors[index]
        .measurementType).lossyScale.x / 2f);
    transform.rotation =
       Quaternion.FromToRotation(Vector3.up, hit.normal);
    transform.localScale = transform.localScale /
       scaledObject;
    return true;
}
else {return false} //Miss
```

Figure 5.11: Algorithm for attaching the sensor to the model surface.

In the event that the `Raycast` does not hit any point of the model, 30 `Raycasts` with different directions are generated. The coordinates of the collision points are calculated for each ray, until the closest point and direction is found, and so proceed with the positioning as previously described. This method is executed only if the first one fails, because the generation of a large number of `Raycasts` has a computational cost, and to obtain the precision provided by `ClosestPoint` it would be necessary to cast a much higher number of rays. If the first method fails, that is because the object is inside a `Collider` (thus failing the `ClosestPoint` function), and the number of rays thrown can be lower because it is close enough to the model.

```
SetMeshConvex(false, meshColliders);
if (!SetSensor(startingPosition, direction))
{
    Vector3[] directions = new Vector3[] {...}
    bool first = true;
    for (int i = 0; i < directions.Length; i++)
    {
        if (first)
        {
            finalHit = hit;
            first = false;
        }
        else
        {
            if (hit.distance < finalHit.distance)
            {
                finalHit = hit;
            }
        }
    }
    direction = (finalHit.point -
       startingPosition).normalized;
    SetSensor(startingPosition, direction);
}
```

Figure 5.12: Second chance algorithm for attaching the sensor to the model surface.

5.7 Monitoring phase

The `StartMonitoring` function contained in the `GameManager` script is called when it is necessary to proceed to the new phase, regardless of the type of connection chosen. It allows to deactivate the interface of the previous phase in order to show the monitoring phase elements:

- the 3D model is placed in a new location, which is at a distance of 2 meters from the user, and is enlarged to allow greater visibility. It keeps the same rotation chosen during the positioning phase of the sensors;
- the panel with the graph and buttons, that it always follows the user's gaze.

To reduce computational cost, the 3D model is simplified through the use of the `CombineMeshes` function, which is in charge of merging into a single mesh.

```
MeshFilter[] meshFilters =
   ModelStructure.GetComponentsInChildren<MeshFilter>();
Mesh finalMesh = new Mesh();
CombineInstance[] combiners = new
   CombineInstance[meshFilters.Length];
for (int i = 0; i < meshFilters.Length; i++)
{
    if (meshFilters[i].transform == transform) continue;
    combiners[i].subMeshIndex = 0;
    combiners[i].mesh = meshFilters[i].sharedMesh;
    combiners[i].transform =
       meshFilters[i].transform.localToWorldMatrix;
}
finalMesh.CombineMeshes(combiners);
ModelStructure.GetComponent<MeshFilter>().mesh = finalMesh;
```

Figure 5.13: Algorithm to combine meshes.

Depending on the connection type, `StartMonitoring` change its flow by activating three different functions:

- `StartWatch` is called for the connection with the database in real-time;
- `FirstBatchData` is called for the non-real-time connection;
- `Send(TypeMessage.RequestDataStart)` is called for the TCP connection.

Regardless of the type of connection, the received data are updated thanks to the `UpdateData` function. It receives as input parameters:

- a dictionary that contains all the sensors (with the wavelength value);
- an array that contains the intensity and radius values for each sensor;
- an optional parameter that represents the latency, available only in the two real-time cases.

This function activates the methods contained in the `GUIManager` script which are responsible for updating the heat-mapped model and the graph: `UpdateShader` and `UpdateCategory`.

5.7.1 Shader-based Heat-map

The Viewer should provide an overview of action clearly and concisely. This is done by using different colors to dynamically display measured value changes during a flight. As already anticipated, each sensor is related with a single heat-map that shows the intensity of the wavelength values compared to the idle values, increasing the intensity of the colors when the absolute value of the delta became bigger. To explain the implementation method, it is important to first introduce how meshes, materials and shaders work in Unity.

Shader overview

A polygon mesh is a collection of vertices, edges and faces, which defines the object in space. Each model contains a set of vertices (represented by coordinates) grouped into three, thus forming a set of triangles. A material is a component associated with the mesh that is used by Unity to render the object itself. The behavior of the material is defined by a shader: a portion of code that contains information and properties that allow its customization. Shaders are composed of a set of instructions, executed altogether for each pixel of the object. The shaders code is executed by the GPU, to take advantage of the high parallel computing capabilities, avoiding the overload for the CPU. The high level of parallelization allows fast execution, allowing the manipulation of the surface of an object in real-time [36].

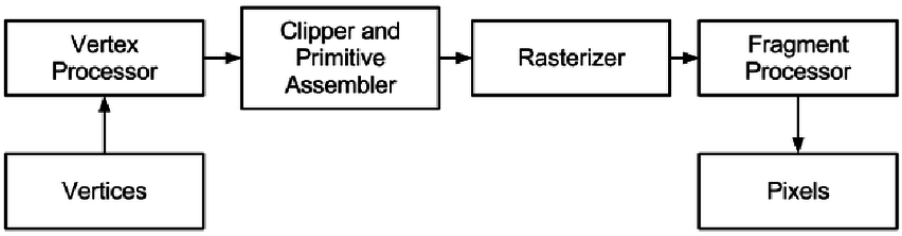


Figure 5.14: Rendering pipeline [37].

The programming language used to customize the shaders is similar to the `C` language and it is later converted to a native shader such as GLSL. Unity shaders are divided into two parts: properties, which are the input values, and sub-shaders which are the part where the code that guarantees the behavior is present. The second part consists of *vertex* and *fragment* programs. The first receives as input the struct `VertexInput`, which holds raw data correlated to position, normal and texture coordinates of the current vertex. The raw data is transformed to be used by the program in world coordinates by the `vert` function, and is saved in the struct `VertOutput`. It is the input parameter for the fragment shader, which is the piece of code that manages the behavior of the heat-map. As introduced in

Section 4.2.1, to allow heat-map custom shader to work in single-pass instanced stereo rendering mode, it was necessary to make changes to these structures. As also shown in Figure 5.15, new instructions have been added in the struct `VertexInput`, `VertOutput` and in the `vert` function.

```
struct VertexInput
{
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float2 texcoord : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID //Inserted
};
struct vertOutput
{
    float4 pos : POSITION;
    fixed4 diff : COLOR0;
    fixed3 worldPos : TEXCOORD1;
    fixed2 uv : TEXCOORD0;
    UNITY_VERTEX_OUTPUT_STEREO //Inserted
};
vertOutput vert(VertexInput input)
{
    vertOutput o;
    UNITY_SETUP_INSTANCE_ID(input); //Inserted
    UNITY_INITIALIZE_OUTPUT(vertOutput, o); //Inserted
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o); //Inserted
    ...
};
```

Figure 5.15: Custom shader with support to single-pass instanced stereo rendering.

Heat-mapped model

The custom shader for the creation of the heat-maps receives as input the position of the sensors in the world space, and their properties divided into radius, intensity and type. As shown in Figure 5.16, the algorithm iterates for each pixel of the model to find the sensor closest to the ith pixel.

```
half4 frag(vertOutput output) : SV_TARGET
{
    ...
    for (int i = 0; i < _Points_Length; i++)
    {
        half ri = _Properties[i].x;
        half di = distance(output.worldPos, _Points[i].xyz);
        half hi = 1 - saturate(di / ri);
        if (hi != 0) near = true;
        if (di < minDistance && near)
        {
                minDistance = di;
            h = hi / 2.0 + _Properties[i].y / 2.0;
            type = _Type[i];
                }
    }
    ...
}
```

Figure 5.16: Distance from sensors calculation.

If the pixel is in range of the nearest sensor, it evaluates the sensor type and, based on that, calls the `getHeatForPixel` function with a certain color array (Figure 5.17). This last function has the task of returning the color contribution given by the intensity received and the proximity of the pixel to the sensor, according to a weighted ratio. The shader structure was inspired by the previous implementation and this tutorial [38].

Before starting the monitoring phase, the `SetSensorHeatmapPositionAndType` function is in charge of setting in the shader code the type of all the sensors to be monitored, and the `UpdateSensorHeatmapPosition` function sets the world position of all the sensors in the `_Points` array.

At runtime, when it is time to update the data, the `UpdateShader` function updates the contents of the `_Properties` array with the intensity and radius values for each sensor using the `SetVectorArray` function. In this way, the shader is constantly updated whenever the parameters passed are different.

Since the model follows the user's view, it is also necessary to update the world coordinates of each sensor in real-time This operation is performed by the `UpdateSensorHeatmapPosition` function, only if the position of the model has changed. This validation is done with the `Update` function located in the `MonitoringModel` script, which checks the `Transform.hasChanged` property to detect if its position has been updated.

```
float3 getHeatForPixel(float h, float4 colors[4])
{
    if (h <= _pointranges[0]) return colors[0];
    if (h >= _pointranges[3]) return colors[3];
    for (int i = 1; i < 4; i++) {
         if (h < _pointranges[i]) {
            float dist_from_lower_point =
               h - _pointranges[i-1];
            float size_of_point_range =
               _pointranges[i] - _pointranges[i-1];
            float ratio_over_lower_point =
               dist_from_lower_point / size_of_point_range;
            float3 color_range = colors[i] - colors[i-1];
            float3 color_contribution =
               color_range * ratio_over_lower_point;
            float3 new_color =
               colors[i-1] + color_contribution;
            return new_color;
        }
      }
    return _tempColors[0];
}
```

Figure 5.17: Get color contribution.

5.7.2 Graph

The graph management is handled by the `GUIManager` class. There are three main functions that manage data visualization:

- `SetGraph`;
- `InsertCategory`;
- `UpdateCategory`.

The first two functions are in charge of initializing the graph, because the plug-in allows it to be completely customized.

`SetGraph` sets values such as the number of divisions, subdivisions, and labels present in the axes.

`InsertCategory` associates a category for a sensor, so that each category corresponds to a line in the graph. More details on how the plug-in works can be found in its documentation [30].

Whenever the data needs to be updated, `UpdateCategory` displays the latency value and calls `AddPointToCategoryRealtime` to update the wavelength values for each sensor. Since the data to be updated increase over time, it is necessary that the method starts counting how many points there are in the graph. When it reaches a set value, the visible values become fixed and the graph becomes a scrollable view. If this operation were not performed, the memory would constantly increase, while the frame rate could no longer maintain acceptable values.

The flight data showed that the delta of the wavelength values never exceeds 1 ns. Due to this reason and to make the graph easier to read, for each line associated with a sensor the scale values oscillate between `-1` and `+1`.

5.8 MeasurementLog

During the development of the application, it was necessary to perform tests for measuring of FPS and memory used by the program. For this reason a class, called `MeasurementLog`, was created. It is for development purpose only and is activated only in debug phase, so that in the release version it does not work. During the execution, three `.csv` files are created:

- `simulation_log_total` collects the data during the entire life of the software;
- `simulation_log_monitoring_phase` collects the data during the positioning phase of the sensors;
- `simulation_log_sensor_position_phase` collects the data during the monitoring phase.

To detect these values the Profiling class provided by Unity is used.

Chapter 6

Tests

At the end of the development of the application, it was necessary to make measurements of its performance and usability. The data were collected in two different ways:

- as described previously in Section 5.8, the first way was to create a script, called `MeasurementLog`, that writes the values related to FPS and memory used in a `.csv` file;
- The second way consisted in collecting CPU and GPU percentage of use through the performance tracking performed by the *Windows Device Portal*, a web server present in HoloLens accessible from common web browsers. It generates an `.etl` files containing log information.

Several tests were performed, all with the same evaluation criteria, but in different contexts: most of the analyses were done using the interrogator emulator, due to the limited access to the *SmartScan* interrogator. It was also the only way to test performance with a high number of sensors. Real tests were also performed in the laboratory using the model aircraft created by the *ICARUS* team.

6.1 Case study: Physical systems

The most interesting and noteworthy experiment was performed with the real interrogator by measuring the value of the sensors placed in the wing and tail of the *Anubi* model aircraft created by the *ICARUS* team. The sensors were located inside the aircraft structure, so it was difficult to recreate the conditions achievable in a normal flight. By recurring to a small carbon strip with two FBG sensors, respectively measuring displacement and temperature, it was possible to perform a small test with the aim to display greater variations. By bending and heating this small piece, it was possible to measure large variations, highlighted by the application during the monitoring phase.

The tests were performed only using the database connection for logistical reasons. The mean latency time was approximately 1 second in each test. The other analyzed parameters, as shown in Table 6.1, were always similar in all cases except for the frame rate value during the first case. As expected, the performances turned out to be acceptable.

# Sensors	Simulation period	CPU	Memory	GPU	FPS
6	13 min 30 s	10.85%	73.16 MB	50%	29.6
16	13 min 11 s	20.05%	72.95 MB	53%	57.6
11	11 min 30 s	17.52%	71.17 MB	55%	59.6

Table 6.1: Overall performance in real-time with physical system.

At the end of the test, the application proved to be useful to discover that the fibers mounted in the wing were damaged, probably during the last flight. In the following subsections the collected data are described in greater detail.

(a) SmartScan Interrogator. (b) Anubi's wing.

Figure 6.1: Wing test.

6.1.1 Test with the wing of Anubi

The first test was performed with a wing of Anubi, in which 6 FBG sensors were mounted. All fibers were placed in the gratings of one single channel, so the other three available channels of the interrogator remained unused. The test duration was 13 minutes, divided into 2 minutes during the configuration phase and 11 minutes in the monitoring phase. As shown in Figure 6.2, CPU usage is quite constant over time, with a usage percentage of 10.85%.

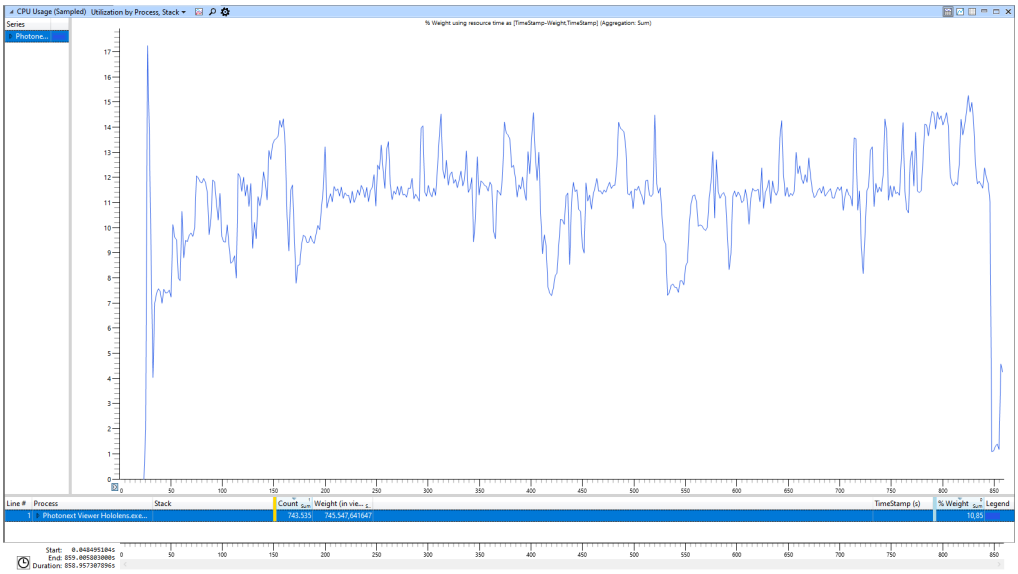
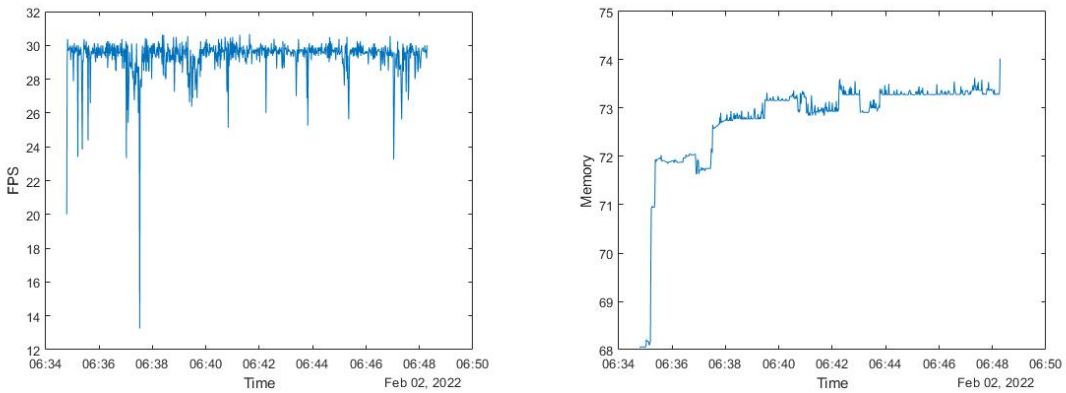


Figure 6.2: CPU usage captured during the test with the wing of Anubi.

The memory used was on average 73 MB, while the frame rate remained constant with some drops on 28/29 FPS. There were no visual problems or slowdowns, although some FPS drops were recorded in `.csv` file. This was due to the fact that the Hololens was also recording a video of the entire testing process. This task is quite expensive and has halved the amount of FPS.



(a) Frame per second.

(b) Memory usage.

Figure 6.3: Performance captured during the test with the wing of Anubi.

	Total	Configuration phase	Monitoring phase
Simulation period	13 min 30 s	2 min 9 s	10 min 46 s
Memory	73.16 MB	71.90 MB	73.27 MB
FPS	29.6	29.5	29.6

Table 6.2: Detailed real-time performance during the test with the wing of Anubi.

6.1.2 Test with the tail of Anubi

The second test was performed with the tail of Anubi, in which 16 FBG sensors were mounted. Differently from the previous case, its fibers were homogeneously arranged in each channel. The duration was similar to the previous one, with a total period of 13 minutes. The CPU usage was slightly higher, 20.05%, while GPU usage was on average 53%.

	Total	Configuration phase	Monitoring phase
Simulation period	13 min 11 s	3 min 2 s	10 min 3 s
Memory	72.95 MB	70.87 MB	73.27 MB
FPS	57.6	58.4	57.1

Table 6.3: Detailed real-time performance during the test with the tail of Anubi.

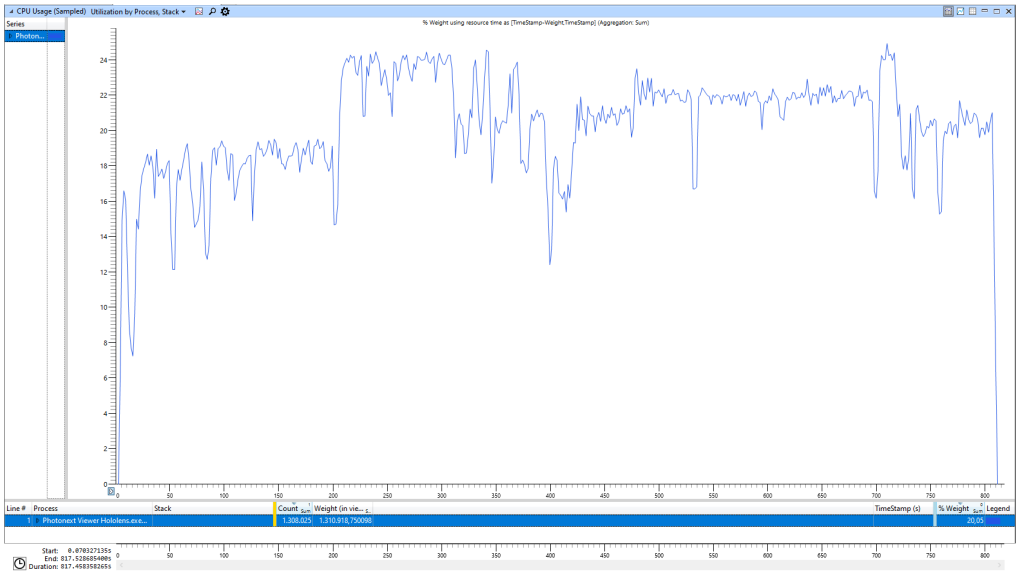
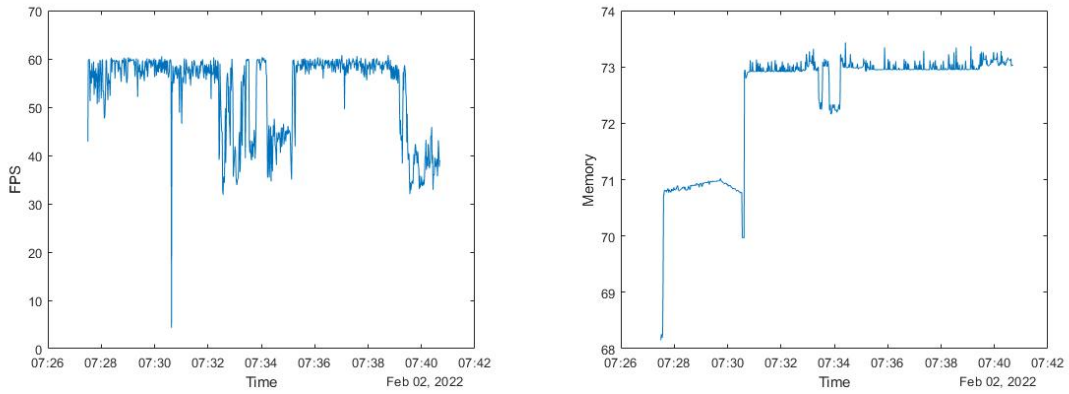


Figure 6.4: CPU usage captured during the test with the tail of Anubi.

As shown in Figure 6.5, while the number of sensors was doubled with respect to the wing, the memory usage remained constant and the refresh rate, with no recording in progress from the headset, went up to 60 FPS.



(a) Frame per second. (b) Memory usage.

Figure 6.5: Performance captured during the test with the tail of Anubi.

(a) Monitoring phase in HoloLens Viewer. (b) Wing and Tail of Anubi.

Figure 6.6: Physical system analysis.

6.1.3 Test with both the wing and the tail of Anubi

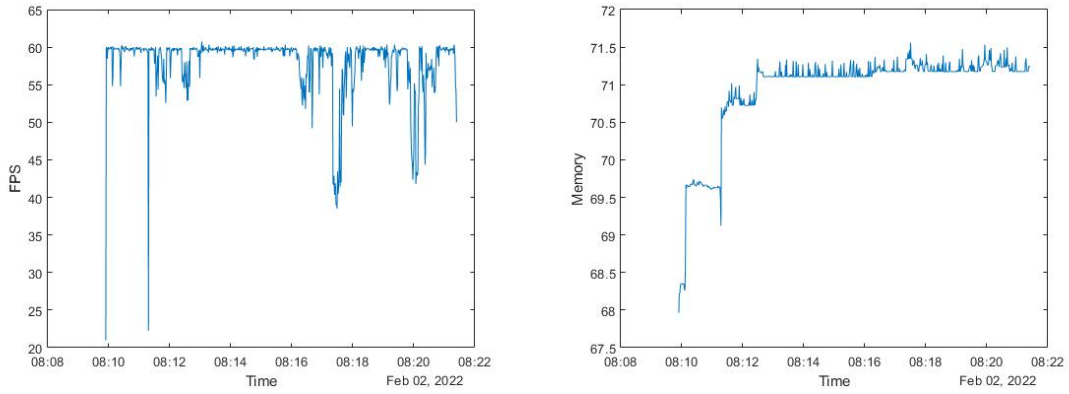
	Total	Configuration phase	Monitoring phase
Simulation period	11 min 30 s	1 min 10 s	10 min 6 s
Memory	71.17 MB	69.65 MB	71.17 MB
FPS	59.6	59.7	59.6

Table 6.4: Detailed real-time performance with the wing and the tail of Anubi.

During the last laboratory test, it was decided to use both the wing and the tail and to divide the channels by the fibers, in which one was coming from the wing and two from the tail. As shown in Table 6.4, simulation time was shorter, about 11 minutes, but the performances were always consistent. The CPU showed a lower utilization rate than the previous case, scoring a value of 17.52%, while the GPU was stable on a 55% usage rate.

Figure 6.7: CPU usage captured during the test with both the wing and the tail of Anubi.

The monitor refresh rate and memory used values remain the same from the previous test, as shown in Figure 6.8.



(a) Frame per second. (b) Memory usage.

Figure 6.8: Performance captured during the test with the wing and the tail of Anubi.

6.2 Case study: Emulator

The final part of the testing was conducted using interrogator emulation software. To verify the behavior of the program, especially with a large number of sensors, several tests were performed using different numbers of active sensors coming from the middleware. Tests with connection to MongoDB and TCP packets were performed. The application was tested in real-time and non real-time mode, using the same collection created during the corresponding real-time analysis with the connection to MongoDB.

6.2.1 Real-time test with MongoDB connection

Thanks to the latest version of the middleware, it was possible to perform many tests using the connection to MongoDB, analyzing the performance in three different cases. The software was first analyzed with 2 sensors (active for 23 minutes), then with 8 (active for 17 minutes), and finally with 36 (active for 13 minutes), simulating the average times of a real flight. While the first two cases are more similar to the real case, the last was used as an extreme case, with triple the actual number of sensors placed on Anubi. As can be seen from Table 6.5, in the first two cases the values are similar, showing good results. On the other hand, if the number of sensors is much higher, performance begins to degrade even if it remains within the threshold of acceptable values.

# Sensors	Simulation period	CPU	Memory	GPU	FPS
2	23 min 13 s	16.20%	69.72 MB	45%	59.8
8	17 min 50 s	19.01%	70.97 MB	46%	59.7
36	13 min 34 s	20.46%	80.59 MB	42%	32.3

Table 6.5: Overall performance during the real-time test with MongoDB connection.

2-sensor configuration test

The first test was conducted by monitoring only 2 active sensors. The test proceeded for 23 minutes, divided into 1 minute for the sensor positioning phase and 22 minutes for the monitoring phase. The number of samples captured was 12680, subsequently grouped by milliseconds to obtain 1394 samples. The CPU and GPU values were very similar to the real system cases previously described: a CPU usage of 16.2% and a GPU usage of 45% were detected. During the monitoring phase the refresh rate remained constant at 60 FPS, with some drops to 38 FPS detected in csv file, during the monitoring phase.

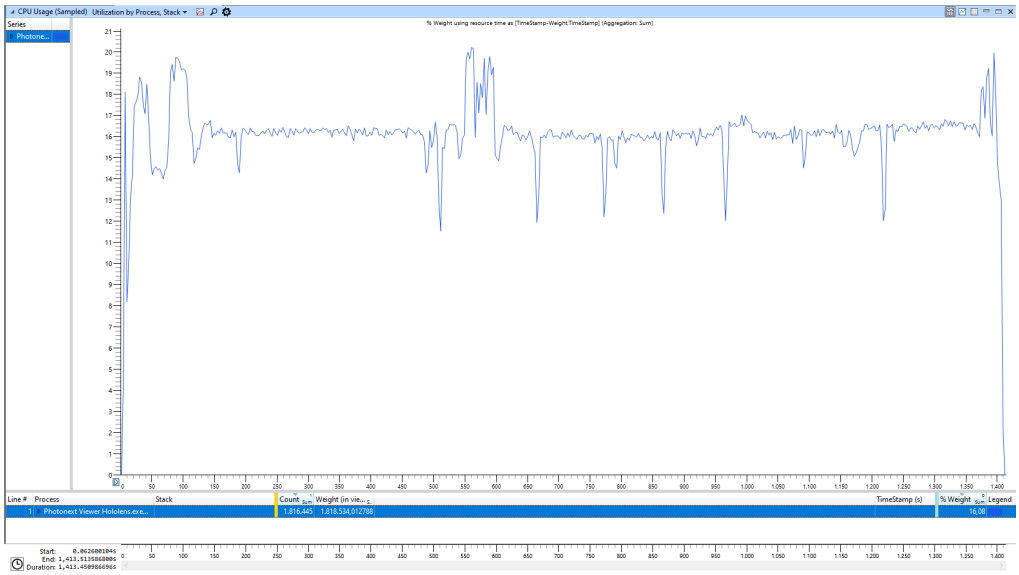
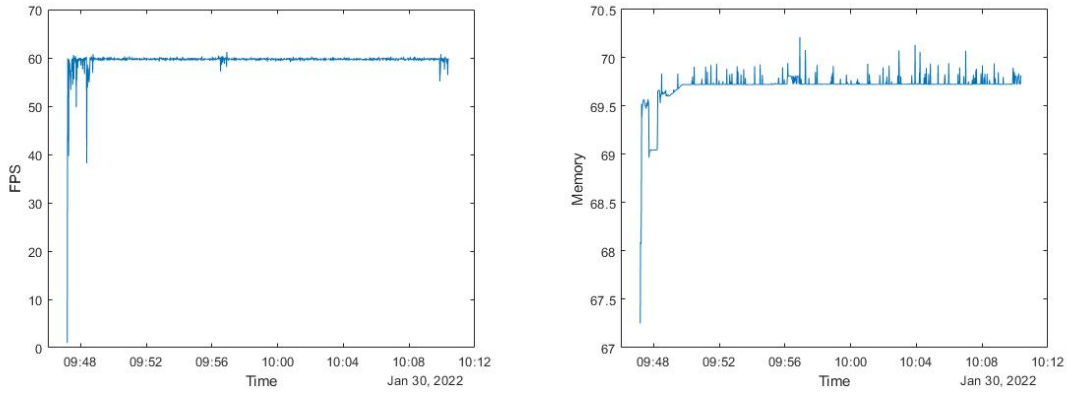


Figure 6.9: CPU usage captured during the 2-sensor configuration test.

Memory usage in both phases averaged 69.72 MB, with a peak during the monitoring phase, hitting 70.22 MB. In general, the experiment using 2 sensors gave the best results.



(a) Frame per second.

(b) Memory usage.

Figure 6.10: Performance captured during the 2-sensor configuration test.

8-sensor configuration test

The following test was performed with 8 sensors. The duration was less than the previous one, about 2 minutes for the sensor positioning phase and 16 for the

monitoring phase, in total almost 18 minutes. The number of samples captured was 9675, subsequently grouped by milliseconds to obtain 1071 samples. CPU and GPU usage rates are always similar to previous results.

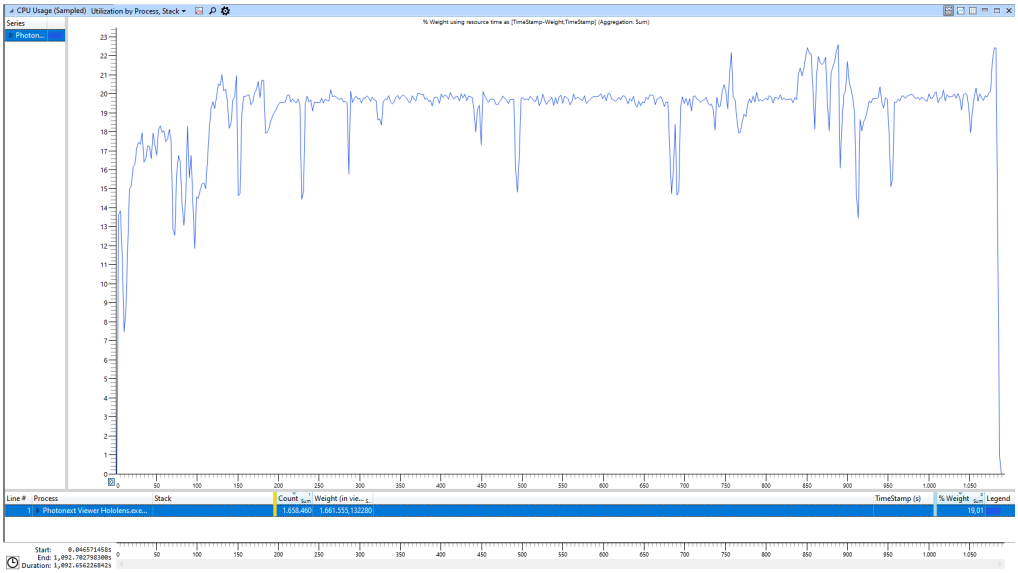
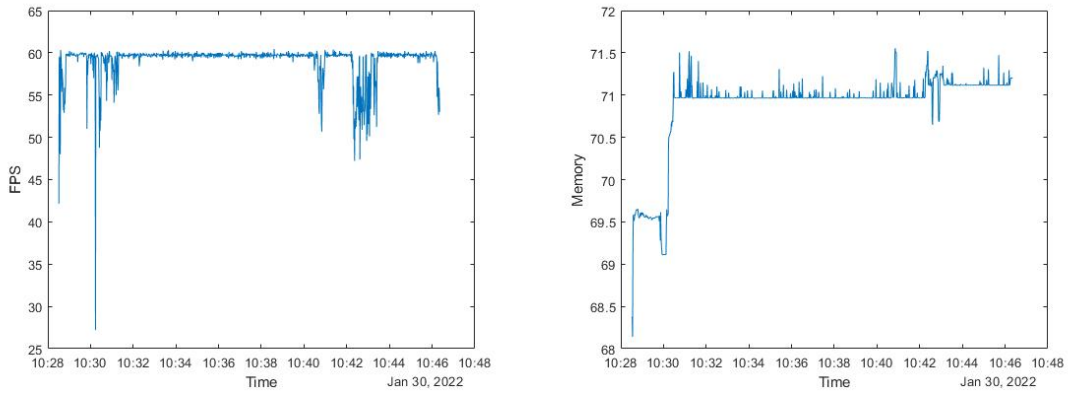


Figure 6.11: CPU usage captured during the 8-sensor configuration test.

Memory usage slightly increased, with a peak recorded during the monitoring phase of 71.55 MB. The frame rate is constant at 60 FPS, although some drops to 50 FPS were recorded towards the end of the simulation, as shown in Figure 6.12.



(a) Frame per second. (b) Memory usage.

Figure 6.12: Performance captured during the 8-sensor configuration test.

36-sensor configuration test

The test conducted with 36 sensors showed the hardware limitations of the HoloLens; in fact, all the measured values were higher than the other cases. As depicted in Figure 6.13, the CPU has reached an average utilization percentage of 20.5%, but with a constant peak of 24% during the monitoring phase.

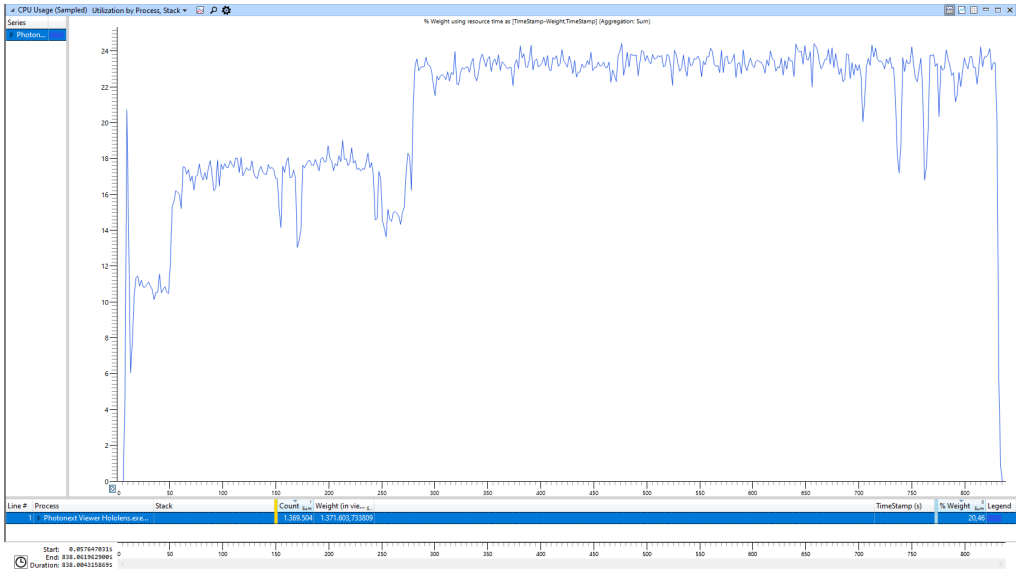
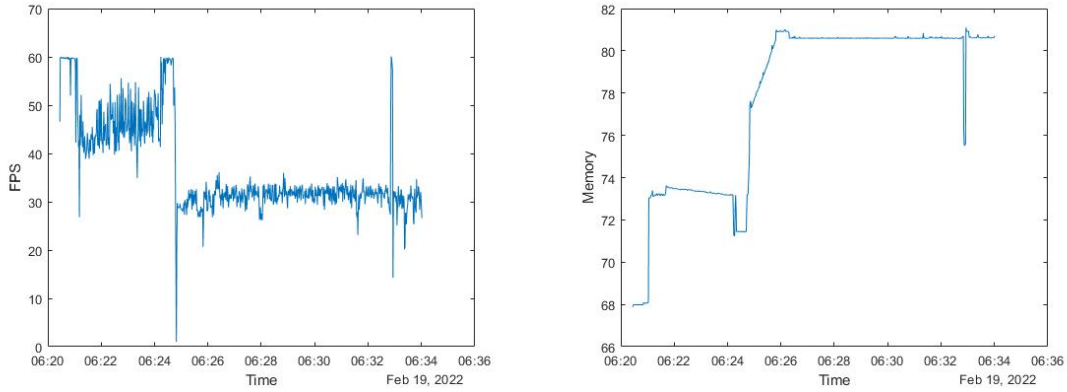


Figure 6.13: CPU usage captured during the 36-sensor configuration test.

The most remarkable data, however, is the refresh rate: with too many sensors, an average of 30 FPS was detected. Although it is an extremely acceptable value, when compared with the other cases there is a big difference. Memory usage also went up, capturing an average of 80.59 MB.



(a) Frame per second.

(b) Memory usage.

Figure 6.14: Performance captured during the 36-sensor configuration test.

6.2.2 Real-time test with TCP connection

Due to the fact that the middleware working with TCP packets was set not to have more than 8 active sensors, the tests with the TCP connection were only two. As can be seen from Table 6.6, the values stay consistent with previous tests.

# Sensors	Simulation period	CPU	Memory	GPU	FPS
2	10 min 10 s	14.44%	70.38 MB	39%	59.7
8	10 min 15 s	17.84%	74.38 MB	43%	57.8

Table 6.6: Overall performance during the real-time test with TCP connection.

2-sensor configuration test

The first test with the TCP connection was conducted by monitoring only 2 sensors, and as can be seen in Figue 6.15, the CPU usage is very low, 14.44%. The test was divided into 1 minute for the positioning phase and 9 minutes for the monitoring phase, for a total of about 10 minutes. The number of samples captured was 5523, subsequently grouped by milliseconds to obtain 611 samples.

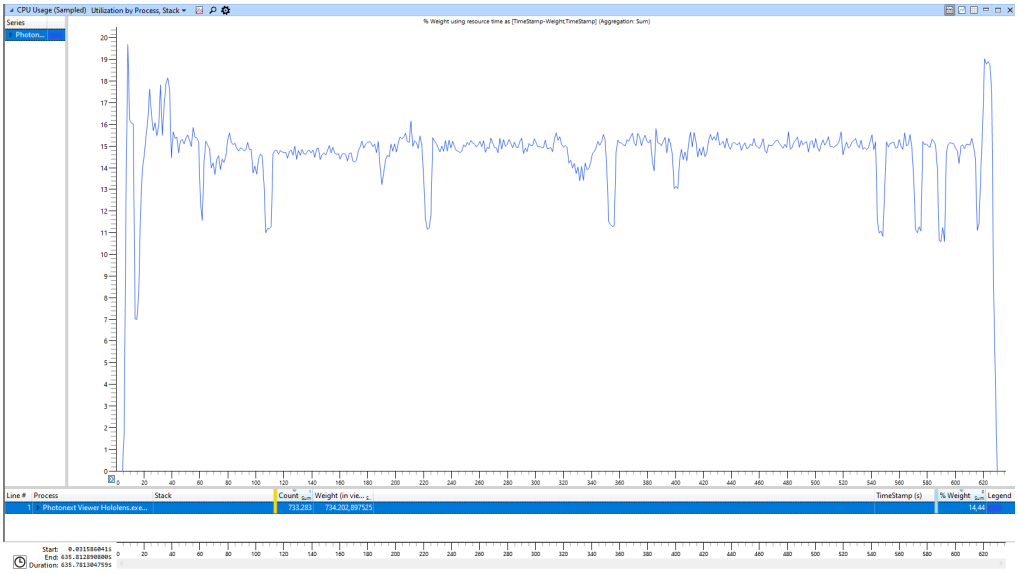
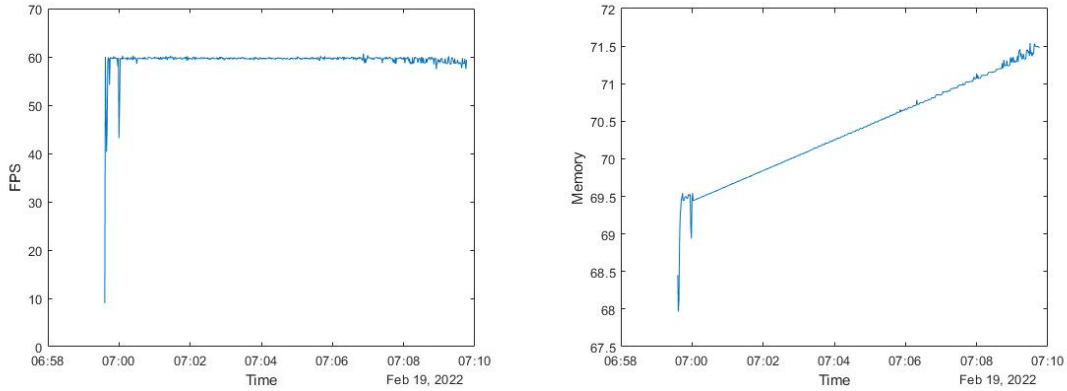


Figure 6.15: CPU usage captured during the 2-sensor configuration test.

Figure 6.16 shows how refresh rates are quite stable at 60 FPS, while memory usage grows to a peak of 71.5 MB. This behavior occurs because the number of samples is still low, and the graph still stores all the values in memory.



(a) Frame per second. (b) Memory usage.

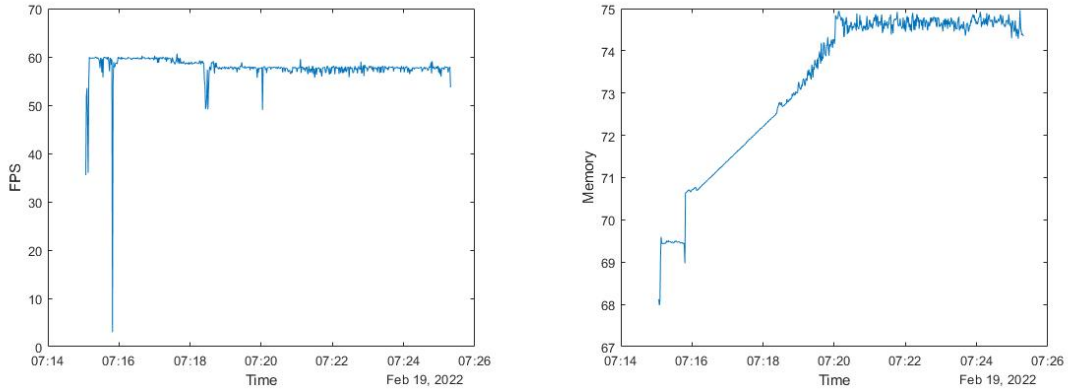
Figure 6.16: Performance captured during the 2-sensor configuration test.

8-sensor configuration test

The test with 8 active sensors was very similar in performance to the previous one. The duration was very similar, 10 minutes total. As shown in Figure 6.17, since the number of monitored sensors increased, also CPU and GPU usage slightly increased.

Figure 6.17: CPU usage captured during the 8-sensor configuration test.

The frame rate captured by the .csv file is also in this case fully acceptable, with an average of 59.7 FPS. As shown in Figure 6.18, the memory usage grows to keep all the samples in the graph, until it remains constant towards 74.38 MB.



(a) Frame per second. (b) Memory usage.

Figure 6.18: Performance captured during the 8-sensor configuration test.

6.2.3 Non real-time

Non-real-time analyses were conducted to simulate the same behavior as testing with the database. For this reason, the number of sensors tested was 2, 8 and 36, using the databases created during the analysis in real time. The duration of all three tests was consistent with the corresponding real-time test and they showed similar behavior overall. Even in the case not in real-time, the program finds it more difficult to monitor a large number of sensors, as shown in Table 6.7. The sensor configurations were reused using the previous saves, for this reason, the sensor positioning phase was skipped in these tests. Tests have shown that the number of database documents saved for each query (10000) is appropriate for running analyses without overloading the hardware too much.

# Sensors	Simulation period	CPU	Memory	GPU	FPS
2	21 min 36 s	16.70%	71.48 MB	43%	57.8
8	15 min 30 s	18.59%	74.31 MB	49%	56.9
36	11 min 39 s	21.66%	78.97 MB	40%	32.6

Table 6.7: Overall performance during the non real-time test with MongoDB connection.

2-sensor configuration test

The first test was conducted by monitoring only 2 active sensors. The test proceeded for 21 minutes. The number of samples captured was 11642, subsequently grouped by milliseconds to obtain 1297 samples. The CPU and GPU values were

very similar to the real-time system cases using MongoDB: a CPU usage of 16.7% and a GPU usage of 43% was detected.

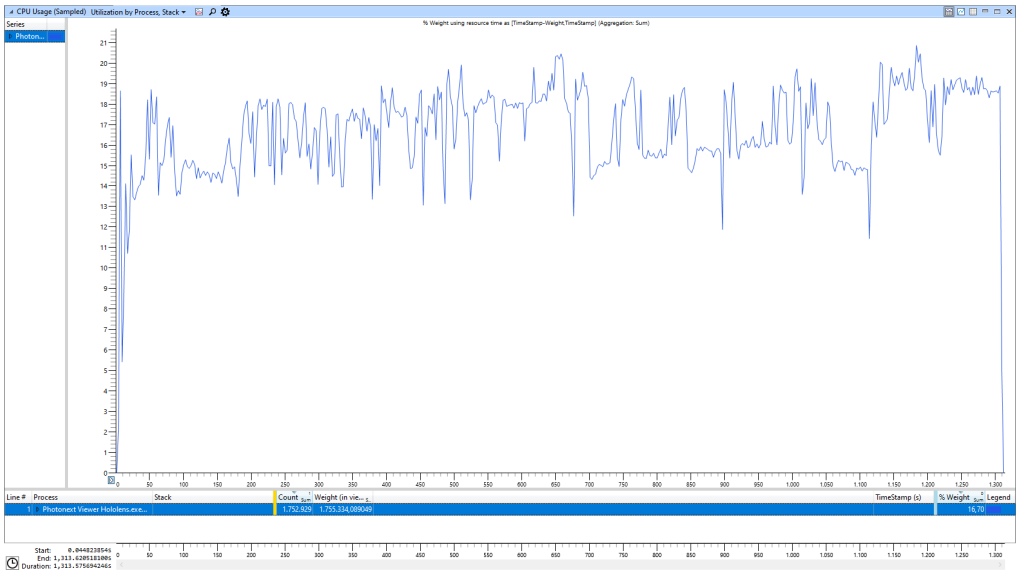
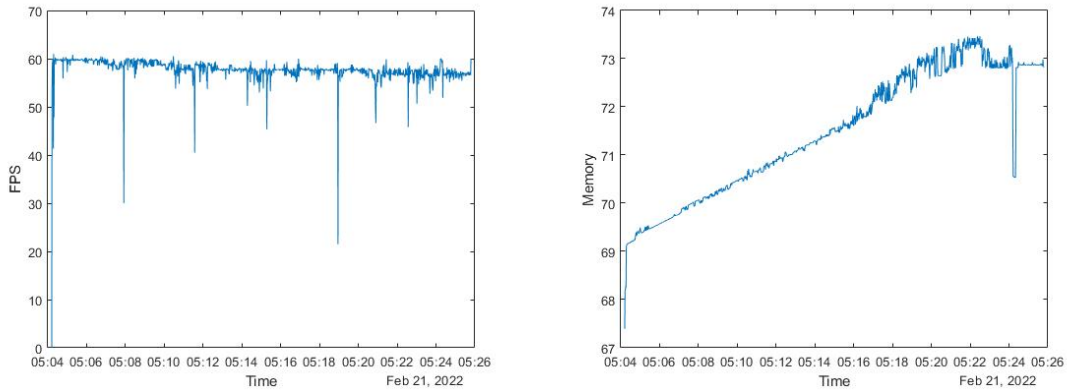


Figure 6.19: CPU usage captured during the 2-sensor configuration test.

The average refresh rate value is 57.8, while the memory usage captured is 71.48 MB.



(a) Frame per second. (b) Memory usage.

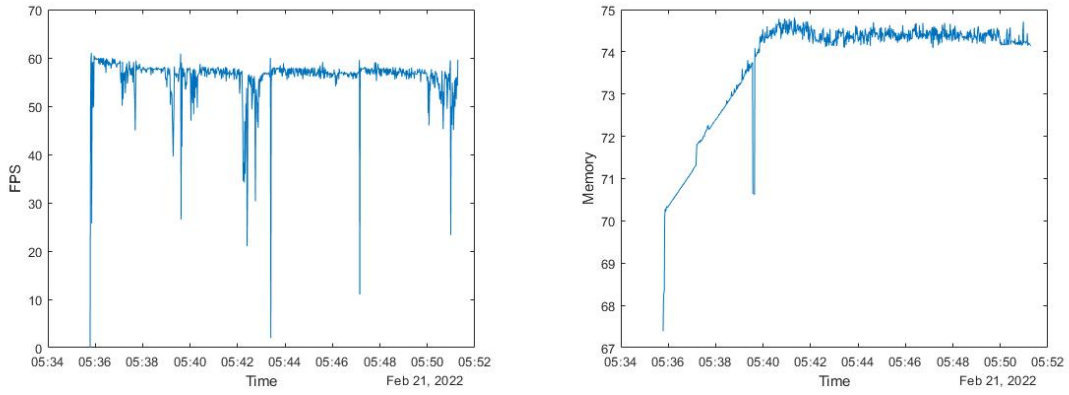
Figure 6.20: Performance captured during the 2-sensor configuration test.

8-sensor configuration test

The following test was performed with 8 sensors. The duration was almost the same to the corresponding real-time analysis, which lasted about 15 minutes. The number of samples captured was 8289, subsequently grouped by milliseconds to obtain 931 samples. CPU and GPU usage rates are similar to the corresponding real-time results.

Figure 6.21: CPU usage captured during the 8-sensor configuration test.

Memory usage and frame rate gave, as expected, the same results.



(a) Frame per second. (b) Memory usage.

Figure 6.22: Performance captured during the 8-sensor configuration test.

36-sensor configuration test

The test conducted with 36 sensors showed the same issues of the corresponding real-time case. CPU and GPU maintain the same average utilization rate.

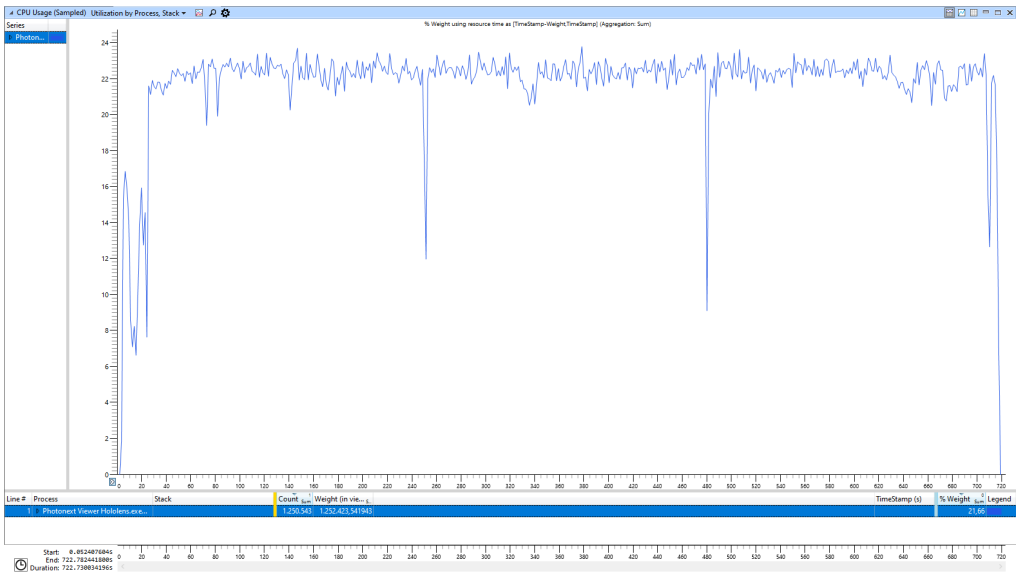
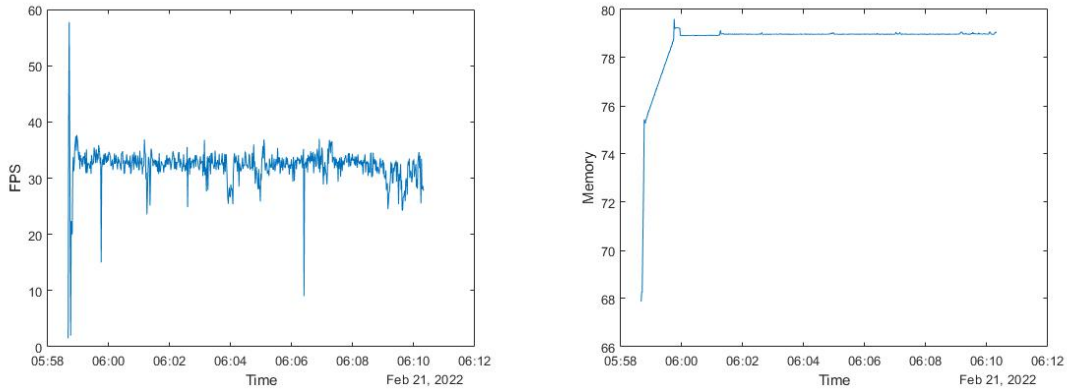


Figure 6.23: CPU usage captured during the 36-sensor configuration test.

The problem encountered even using non real-time data remains the refresh rate. Although it is still acceptable, 32 FPS on average is a big drop compared to the values of the previous tests.



(a) Frame per second.

(b) Memory usage.

Figure 6.24: Performance captured during the 36-sensor configuration test.

6.3 Final analysis

Overall, the results were positive in all cases described. The target frame rate was 30 FPS. Since a stable refresh rate of 60 FPS has been reached, with drops to 30 only in the presence of a large number of sensors, it is possible to state that the requirement is fully satisfied. In extreme cases, only conducted as tests but not feasible in real conditions, the performances tend to degrade due to the limited calculation capacity of the HoloLens, but still remain acceptable. There were no clear differences between connection types, as all analyzes proved to be similar in performance. Figure 6.25 shows a summary graph of the refresh rate values for all the analyzed cases. It is important to note that, apart from the two cases with 36 sensors, the only case in which the average value of frame per seconds are below the average of 60 is the case in which video recording from the device has been activated during monitoring view.

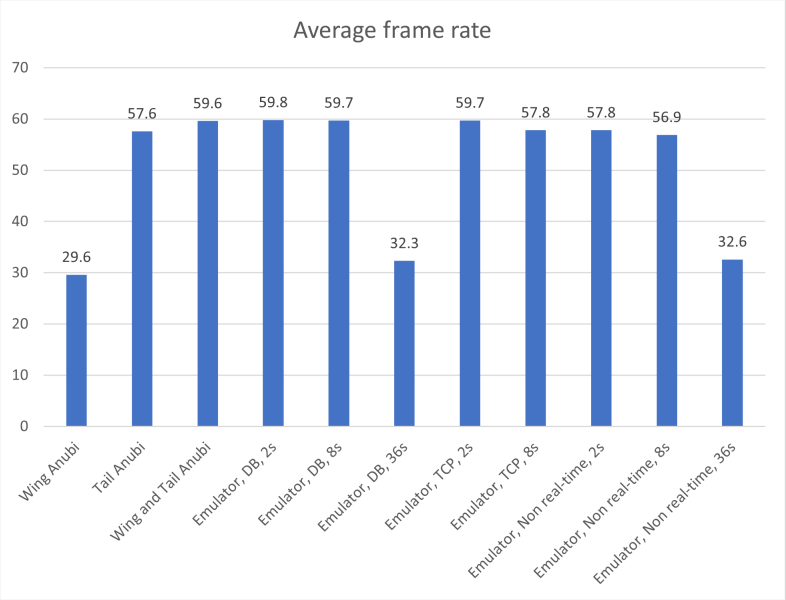


Figure 6.25: Frame rate overview.

HoloLens Viewer was tested by various members of the *ICARUS* team to gather feedback on usability. For the purposes of the tests, it was considered more relevant to have the application tested by few people but from the sector of interest of the application itself. After becoming familiar with the use of the headset, the application was easy and immediate to use. It was much appreciated the way in which the application is configured, and the graph and the heat-mapped 3D model generated general enthusiasm.

Chapter 7

Conclusions

7.1 Results

The aim of this thesis was to enhance the PhotoNext project by creating a reliable AR application capable of displaying data from FBG sensors in real-time and not. The choice to use AR, and in particular *Microsoft HoloLens 2*, was born from the request to add a layer to allow pilots to monitor immediately and effectively the model aircraft status during a flight session. This made it possible to learn technologies and to develop software design with a different approach, especially in terms of input capability. The expected goal was to design a user-friendly software from the ground up that allows the end-user not to waste too much time during sensors configuration phase and to analyze and monitor the situation during a flight session in a quick and functional way. Many aspects were considered during the development phase: starting from the behavior of the FBG sensors and the middleware, and the needs required to create an immediate and easy-to-use application.

The developed application has achieved the objectives requested during the design phase. The conducted tests have shown the behavior of the software in real and extreme cases, remaining in all the scenarios analyzed above the performance threshold given by the initial requirements.

The feedback from people from the sector of interest was positive, and it generated a lot of enthusiasm. Hololens Viewer was considered very intuitive and useful in real-time monitoring, so much that they absolutely want to use it during future flight sessions. The development process made it possible to gain more experience in creating GUI for AR applications and in their programming. It has also provided me with more practical skills, such as expertise in the field of photosensitivity.

7.2 Future works

Although Hololens Viewer is fully functional, the entire PhotoNext project is constantly evolving. The tests, both with the TCP protocol but above all with the database, have shown a weakness that could be improved by using the UDP protocol. Since it is a connectionless type, it does not manage the reordering of packets or the retransmission of lost ones, and is therefore generally considered to be of lower reliability. On the other hand, it is very fast because there is no latency for reordering and retransmission and is efficient for real-time applications.

To implement the UDP protocol it is necessary to intervene both in the AR application, but also in the middleware, which should be set up to send packets of different types.

An improvement of the proposed solution could be to add greater customization of the graphs, while trying to keep the interaction to a minimum base so as not to have distractions during a real-time session. One of the possible additions could be to divide the graph to show, at the user's convenience, only the displacement or temperature values. Also, the plug-in used for building the graphs is not open-source. A possible upgrade could be to develop a new open-source tool from the beginning, so that everyone can use, test and improve it.

Among the future works planned, one of the main ideas is to add to the aircraft an inertial platform, which allows to detect further information during the flight, such as turns and rotations of the model aircraft itself.

Their management can be then added to Hololens Viewer in order to offer the user new visualization modes.

Bibliography

[1] Thiago Tadeu Amici, Peter Hanser Filho, and Alexandre Brincalepe 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)*, pages 1402–1406, 2018.

[2] Microsoft. Hololens 2 overview and technical specifications. `https://www.microsoft.com/en-us/hololens/hardware`.

[3] Politecnico di Torino. PhotoNext. `https://www.photonext.polito.it/`.

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

[5] K.O. Hill and G. Meltz. Fiber bragg grating technology fundamentals and overview. *Journal of Lightwave Technology*, 15(8):1263–1276, 1997.

[6] Antonio Scaldaferri. 3D visualization and analysis of a large amount of real-time and non-real-time data. 04 2021.

[7] Politecnico di Torino. ICARUS. `https://icarus.polito.it/`.

[8] Smatfibres. SmartScan. `https://www.smartfibres.com/products/smartscan`.

[9] Mauro Guerrera. Algorithms and methods for fiber bragg gratings sensor networks. 12 2018.

[10] Ahmad El Zein. Improvement of a system for retrieving and displaying systematic aircraft data. 12 2021.

[11] KaaIoT Technologies. KaaIoT platform. `https://www.kaaiot.com`.

[12] Maria Giulia Canu. Mixed Real-Time Visualization Framework for FGB IoT sensors. 07 2019.

[13] Microsoft. Air-tap gesture. `https://unity.com`.

[14] Microsoft. Start gesture. `https://unity.com`.

[15] Microsoft. Hololens 2 interaction models. `https://docs.microsoft.com/en-us/learn/modules/learn-mrtk-tutorials/1-6-interaction-models`.

[16] Unity. Unity. `https://unity.com`.

[17] Marie Dealessandri. What is the best game engine: is unity right for you? `https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you`, 2020.

[18] Unity.com. Unity solutions. `https://unity.com/solutions`.

[19] Unity Technologies. Unity manual - API documentation. https://docs.unity3d.com/Manual/index.html.

[20] Unity. Unity - Manual XR. https://docs.unity3d.com/Manual/AROverview.html.

[21] Unity. XR Plugin Architecture. https://docs.unity3d.com/Manual/XRPluginArchitecture.html.

[22] Unity. Single-Pass Stereo Rendering for HoloLens. https://docs.unity3d.com/Manual/SinglePassStereoRenderingHoloLens.html.

[23] Microsoft. MRTK. https://docs.microsoft.com/en-us/windows/mixed-reality/mrtk-unity/?view=mrtkunity-2021-05.

[24] Microsoft. MRTK Button. https://docs.microsoft.com/en-us/windows/mixed-reality/mrtk-unity/features/ux-building-blocks/button?view=mrtkunity-2021-05.

[25] Microsoft. MRTK Scrolling Collection. https://docs.microsoft.com/it-it/windows/mixed-reality/mrtk-unity/features/ux-building-blocks/scrolling-object-collection?view=mrtkunity-2021-05.

[26] Microsoft. What is the mixed reality toolkit. https://docs.microsoft.com/en-us/windows/mixed-reality/mrtk-unity/?view=mrtkunity-2021-05#ux-building-blocks.

[27] Microsoft. MixedRealityToolkit-Unity. https://github.com/microsoft/MixedRealityToolkit-Unity.

[28] Unity Asset Store. Runtime OBJ Importer by Dummiesman. https://assetstore.unity.com/packages/tools/modeling/runtime-obj-importer-49547.

[29] Unity Asset Store. Graph and chart by Bitsplash Interactive. https://assetstore.unity.com/packages/tools/gui/graph-and-chart-78488.

[30] Bitsplash Interactive. Graph and chart. http://bitsplash.io/graph-and-chart.

[31] MongoDB. MongoDB. https://www.mongodb.com.

[32] Mayur M Patil, Akkamahadevi Hanni, C H Tejeshwar, and Priyadarshini Patil. A qualitative analysis of the performance of MongoDB vs MySQL database based on insertion and retriewal operations using a web/android application to explore load balancing — Sharding in MongoDB and its advantages. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pages 325–330, 2017.

[33] Diogo Augusto Pereira, Wagner Ourique de Morais, and Edison Pignaton de Freitas. NoSQL real-time database performance comparison. *International Journal of Parallel, Emergent and Distributed Systems*, 33(2):144–156, 2018.

[34] Pim De Witte. UnityMainThreadDispatcher. https://github.com/PimDeWitte/UnityMainThreadDispatcher.

[35] Stefan Bock. MongoDB-IL2CPP. https://github.com/SteepSheep/MongoDB-IL2CPP.

[36] Patricio Gonzalez Vivo and Jen Lowe. The Book of Shaders. https://thebookofshaders.com.

[37] Curran Kelleher. The Rendering Pipeline. https://www.researchgate.net/figure/The-Rendering-Pipeline-Pixars-introduction-of-shaders-in-RenderMan-Upstill-1990_fig1_262398551.

[38] Alan Zucconi. Arrays Shaders in Unity 5.4+. https://www.alanzucconi.com/2016/10/24/arrays-shaders-unity-5-4/, 10 2016.