Bring ideas to life
VIA University College

Android Remote Controller for Mixed Reality Head-Mounted Displays

Using Bluetooth Low Energy

Mario Petrov Burgov 240303

Bogdan-Laurentiu Ene 240298

Bence Sólyom 239842

Supervisor: Kasper Knop Rasmussen

SYSTEMATIC

70469 characters

ICT Engineering

7th semester

12.12.2018

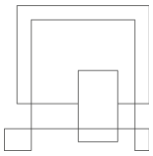
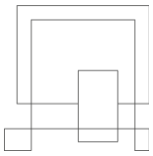
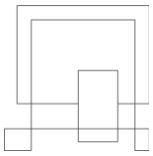


Table of content

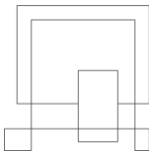


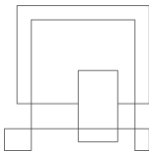


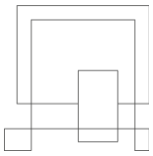
Glossary of acronyms

HMD	Head Mounted Display
AA	Android Application
DTA	Designated Touchpad Area
MR	Mixed Reality
BLE	Bluetooth Low Energy
UWP	Universal Windows Platform

List of figures and tables







1 Abstract

The aim of this project is to create an Android application, that is capable of interacting with and remotely controlling an application running on a mixed reality Head Mounted Display (HMD). The main purpose is to provide a more user-friendly way of interacting with mixed reality HMDs, since current solutions are somewhat rudimentary.

In order to achieve that, the team has come up with the idea of creating a connection between an Android device and an HMD through Bluetooth Low Energy. Through that connection the Android device would send commands to the HMD in the form of data packets, based on user input (finger gestures, button taps, text input) on the Android application.

The exact HMD the team has worked with during the project was Microsoft HoloLens.

The project was also completed in collaboration with the Danish software development company Systematic. The project objective has practical relevance for Systematic, since they have a mixed reality project on Microsoft HoloLens, and its effectiveness could be greatly improved by the remote controller application.

The most notable results of the project include the following:

- *Gesture recognition system in the Android application with a suitable user interface*
- *Sending executable commands in the form of data packets from an Android device to Microsoft HoloLens through Bluetooth Low Energy, based on user input*
- *Integration of Systematic's HMD application into the Android remote controller system*

2 Introduction

The goal of the project is to develop a remote controller system for an MR head-mounted display, providing new ways to interact with MR.

"The SitaWare Suite is a fully integrated range of top-to-bottom C4I (Command, Control, Communications, Computers, & Intelligence) and Battle Management Systems that give you shared situational awareness on all levels of command together with powerful Command & Control tools." (Systematic | Defence, 2018)

Since SitaWare is related to complex operational environments and battle management systems, the existing controls of HMDs are not sufficient, and an improvement is required.

The main problem discovered in SitaWare's HMD implementation are the limited and unreliable controls the Microsoft HoloLens provides.

"Hand gestures do not provide a precise location in space" (Gestures - Mixed Reality | Microsoft Docs, 2018)

Even performing simpler actions (e.g. pressing a button) requires a significant amount of precision from the user, making the applications more difficult to use. Hand gestures might not be recognized properly at times and having the user's gaze as a selection/navigation tool can prove to be rather cumbersome.

This only gets worse in a confined environment, where the user doesn't have a lot of space to maneuver and interact with the system.

In some areas of work, where time is critical, and accuracy of the user's interaction with the system is paramount, this might turn out to be problematic for the user and the outcome of the situation.

For soldiers, that is often the case, as they are required to operate in areas with limited space (e.g. inside a military vehicle or military headquarters), or dangerous situations (e.g. on the frontline of the battlefield).

The idea is to develop an Android application prototype that would improve the current controls and extend them with additional functionalities.

The stakeholder of the project is Systematic, a Danish software development company with 4 departments, with a new project in the Defence department. The project involves the usage of SitaWare through an MR head-mounted display (HMD).

Figure 1 describes how the Android Remote Controller fits in the current SitaWare HMD implementation.

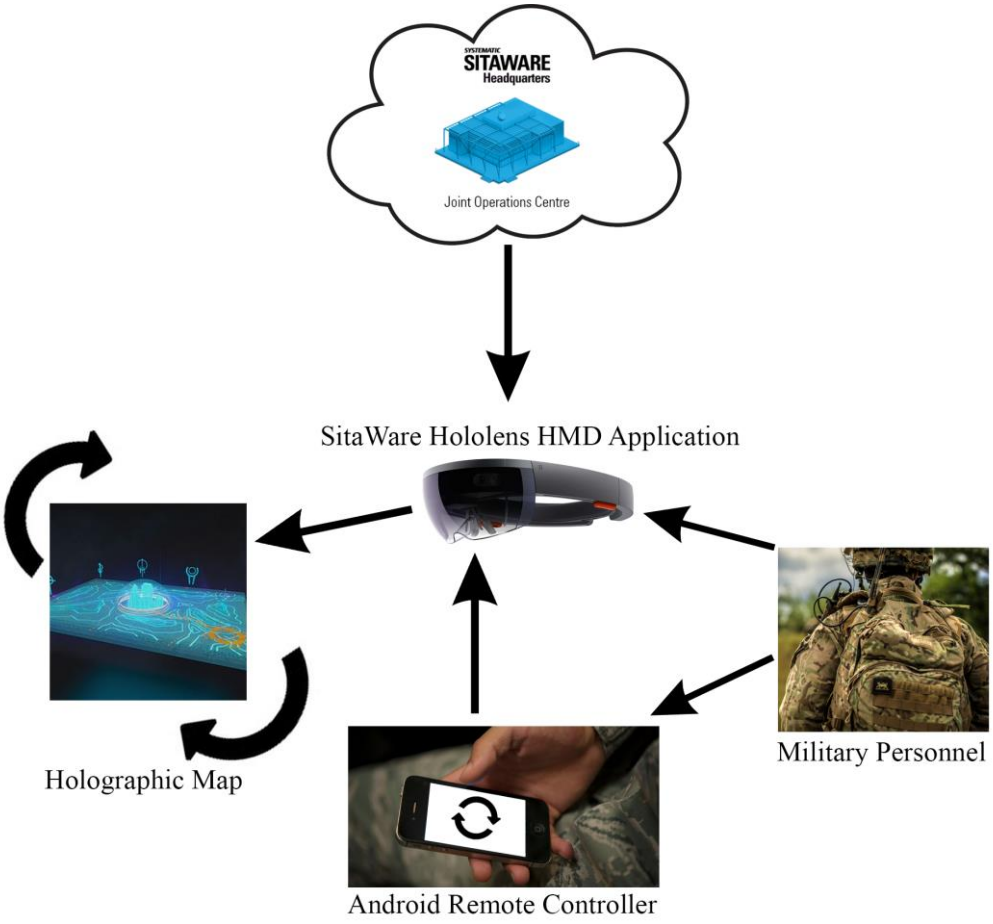
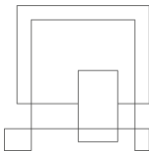


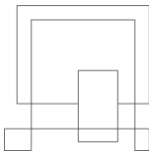
Figure 1 - Rich picture

In the following chapters the system is explained in more details as follows:

- Requirements - This section defines functional and non-functional requirements of the system. The use case diagram and use case descriptions can be found here.



- Analysis - This section outlines an understanding of the problem domain based on the requirements. The background description and the domain model of the system can be found here.
- Design - This section outlines how the system is structured from a development point of view. Descriptions of the architecture of the system and the technologies involved can be found here, along with class diagram, interaction diagrams and the choice of design patterns.
- Implementation - This section describes the development of the system, including, but not limited to implementation of design patterns and a walkthrough from the UI to the "back-end" of the system.
- Test - This section documents the approach and results of testing.
- Results and discussion - This section presents the outcome and achieved results of the project.
- Conclusions - This section compiles the results from each section in the report.
- Project future - This section reflects on the technical viewpoint of the project and suggests how the project could be improved or made ready for production.



3 Requirements

This section defines functional and non-functional requirements of the system. The use case diagram and use case descriptions can be found here.

The following requirements were formulated based on an interview (for more information please refer to Process Report, Chapter 5: Project Execution) between the stakeholders and the development team:

3.1 Functional Requirements

The following requirements were prioritized using the MoSCoW method. (MoSCoW method, 2018)

3.1.1 Must have

Communication between Systematic's HMD app and the AA

Description: The AA should be able to send instructions to the HMD application through a wireless technology. The connection between the AA and HMD application is top priority and the interaction between the two and the SitaWare software would be handled separately.

Rationale: In order to make sure, that the user is able to use the HMD with the assistance of the AA

Panning

Description: The user should be able to pan the map using the arrow keys on the navigation screen

Rationale: In order to provide a more accurate input method for panning the map

Zoom controls

Description: The user should be able to zoom in/out by using the pinching gesture on the DTA

Rationale: In order to make it possible for the user to accurately view the existing map using widely used and intuitive gestures

Rotating

Description: The user should be able to rotate the map using the finger gestures on the DTA

Rationale: In order to make it possible for the user to rotate the map while standing still

Tap selection

Description: The user should be able to use the selection functionality by tapping the DTA in the navigational screen of the AA

Rationale: In some scenarios it would be faster and more reliable than the current MR hand gesture controls

Scaling

Description: The user should be able to scale up or down the map using the settings menu

Rationale: In some scenarios it would be better for the user to change the size of the map in accordance to the surrounding environment

3.1.2 Should have

Keyboard input

Description: The user should be able to input text into the HMD application using a virtual keyboard present on the AA

Rationale: Necessary if the user needs to add text input while searching for maps or adding text to existing map markers

3.2 Non-Functional Requirements

1 Availability

1.1 Keeping the connection persistent

Description: The connection between the system's communication interfaces should persist regardless of the application's state

Rationale: In case something unexpected happens to the AA, the connection between the device and the HMD/system running the HMD application should persist

2 Performance requirements

2.1 Preventing standby mode

Description: The AA should prevent the phone from entering standby mode

Rationale: In order to make sure the user always has access to the AA menu/tools

2.2 Saving latest state

Description: The AA should preserve its latest state before losing focus

Rationale: In case the device loses focus of the AA, the AA should return to the last known state to avoid confusion and save time for the user

3 Usability requirements

3.1 Rotating screen

Description: Changing the layout of the application depending on the orientation of the device. (The AA should be responsive towards the orientation of the device)

Rationale: In order to support different kinds of interaction methods depending on the end-user's preference

4 Interoperability requirements

4.1 Compatibility with different HMD implementations

Description: It should be possible for the AA to be compatible with different HMD implementations with minimal code changes

Rationale: It should save implementation time and money for future software integrations

As a result, from the requirements, the Use Case Diagram in Figure 2 was formulated:

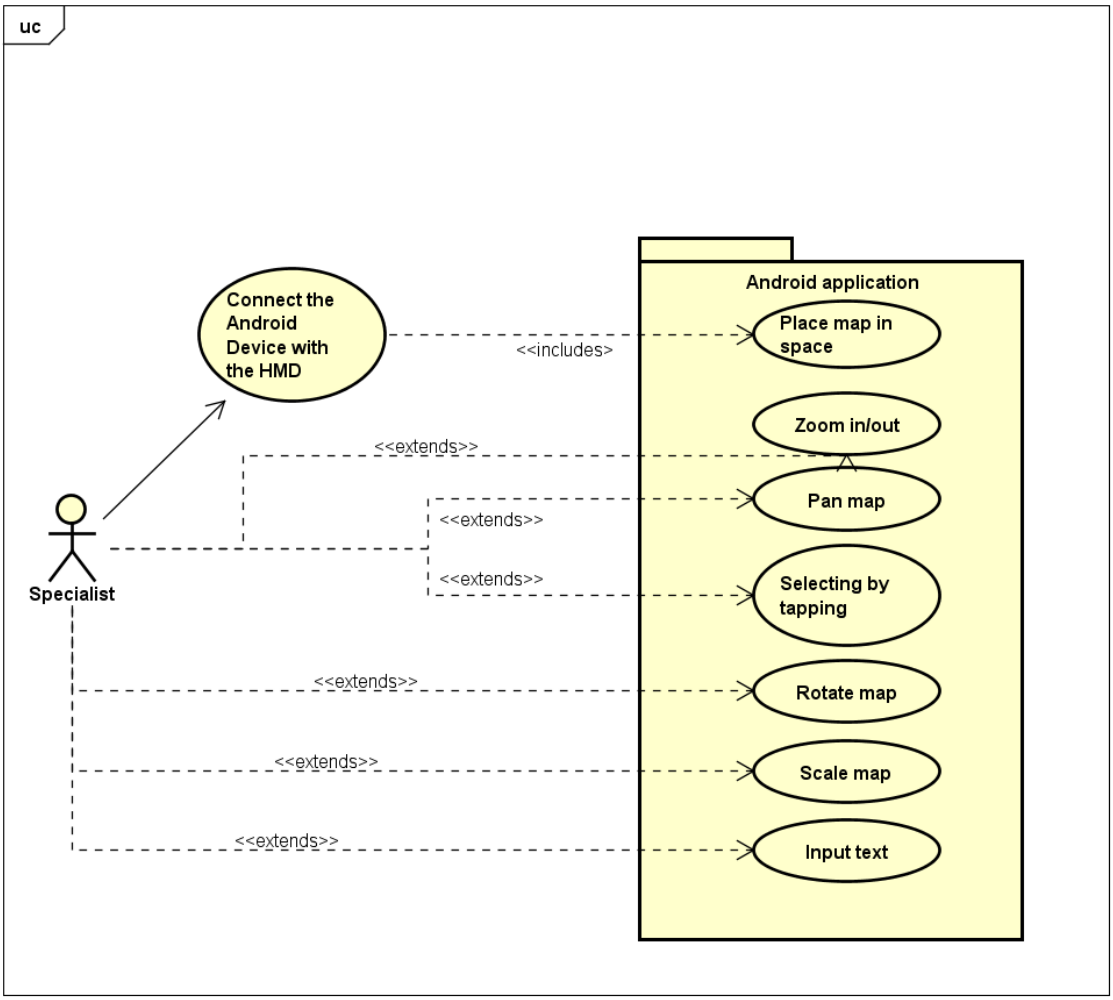


Figure 2 - Use case diagram

In Figure 3, the use case description of the connection between the HMD and AA is displayed as an example. The use case descriptions for the other cases can be found in Appendix B.

ITEM	VALUE
UseCase	Connect the Android Device with the HMD
Summary	The AA should be able to send instructions to the HMD application through a wireless technology. The connection between the AA and HMD application is t op priority and the interaction between the two and the SitaWare software wo uld be handled separately.
Actor	Specialist
Precondition	The AA and HMD should be paired beforehand
Postcondition	
Base Sequence	1. Launch the Android Applciation 2. Select the HMD from the device list 3. Wait for the connection to be enstablished 4. The connection is enstablished
Branch Sequence	
Exception Sequence	In case the connection couldn't be enstablished: Base sequence 1-2-3 4. The user is informed the connection couldn't be enstablished
Sub UseCase	
Note	

Figure 3 - Use case description

4 Analysis

This section outlines an understanding of the problem domain based on the requirements. The background description and the domain model of the system can be found here.

As mentioned before, the SitaWare Suite also has software solutions for Mixed Reality headsets (Microsoft HoloLens). The current implementation of this system consists of displaying a rectangular 3D map (which is placed on a plain surface by the user), showing tactical information received from the central SitaWare server. The information displayed is made up from markers with specified coordinates for units with different affiliations (hostile, friendly, neutral, unknown).

The user is able to interact with the holographic map via HoloLens' built-in pinching gesture or the Microsoft clicker (a small controller with one button) for HoloLens. Pressing the button on the clicker or performing a pinch gesture in front of the HoloLens is registered as a selection command. The user's gaze is recognized by the HoloLens, providing a virtual cursor, indicating the area/object to be interacted with. The actions available for the user are the following:

- Panning the map (by selecting virtual arrow buttons placed around the map)
- Using the current user interface (consisting of 2 pairs of +/- buttons):
    - Zooming in or out of the map (by selecting the corresponding buttons on the UI)
    - Modifying the exaggeration of the terrain in the term of altitude (by selecting the corresponding buttons on the UI)
    - Changing the displayed geographical area (by choosing between predefined coordinates from a drop-down list)

There are several problematic aspects regarding the current controls of the HoloLens. The gaze feature requires a great amount of accuracy from the user, as it is very shaky/inaccurate. This proves to be a hindrance as far as user experience is concerned.

The fact that the user will lose focus of the map in order to access the navigation or the zoom in/out features is also a big disadvantage. The pinching gesture as selection can be exhausting in the long run to perform, not to mention it also has accuracy related issues. The SitaWare system requires precise controls in order to be used efficiently.

Considering everything presented in the above paragraphs, an interview with the stakeholder and a domain expert took place, in which a number of questions were answered, giving the team deeper insight on the overall problem.

After taking the problems into consideration, a brainstorming session for better controls had begun, and found that the most convenient solution would be an Android application that can be used as a remote controller for the HMD implementation.

The backbone of the project consists of creating a connection between the HMD and the AA, in order to achieve the system's main purpose: replacing the current controls with more stable ones. The next step was to get and interpret the user's input. For that, the AA would have a DTA, consisting of a region where the user could control the HMD application by using buttons and existing general smartphone touch gestures.

An interface for the HMD app was created in order to make sure that the AA can be adapted to any suitable system with minimal code changes. In order to help the development and testing processes, a mock-up application was created for the HMD. On this mock-up all the must have functionalities of the AA were tested.

The Domain model consists of the following entities:

- **Soldier -** the user of the system, who might be located in a military headquarters, military vehicle or the battlefield.
- **Server -** the SitaWare server is responsible for providing the displayed maps on the HoloLens.
- **HMD -** consists of Microsoft's HoloLens which runs the current implementation of SitaWare's MR system.
- **Map** - the holographic object representing a geographical area, including tactical information, displayed inside the HMD.
- **Phone** - the Android device running the AA
- **AA** - the remote controller Android Application controlling the application running on the HoloLens remotely

- **DTA** - Designated Touchpad Area, where the user interacts with the map (panning, zooming, rotating the map)
- **Settings** - a submenu containing adjustable user preferences
- **Virtual keyboard** - Android's built-in on-screen keyboard, used to input text to the system

Based on the answers the list of requirements for the new solution and the following domain model presented in Figure 4 was formulated.

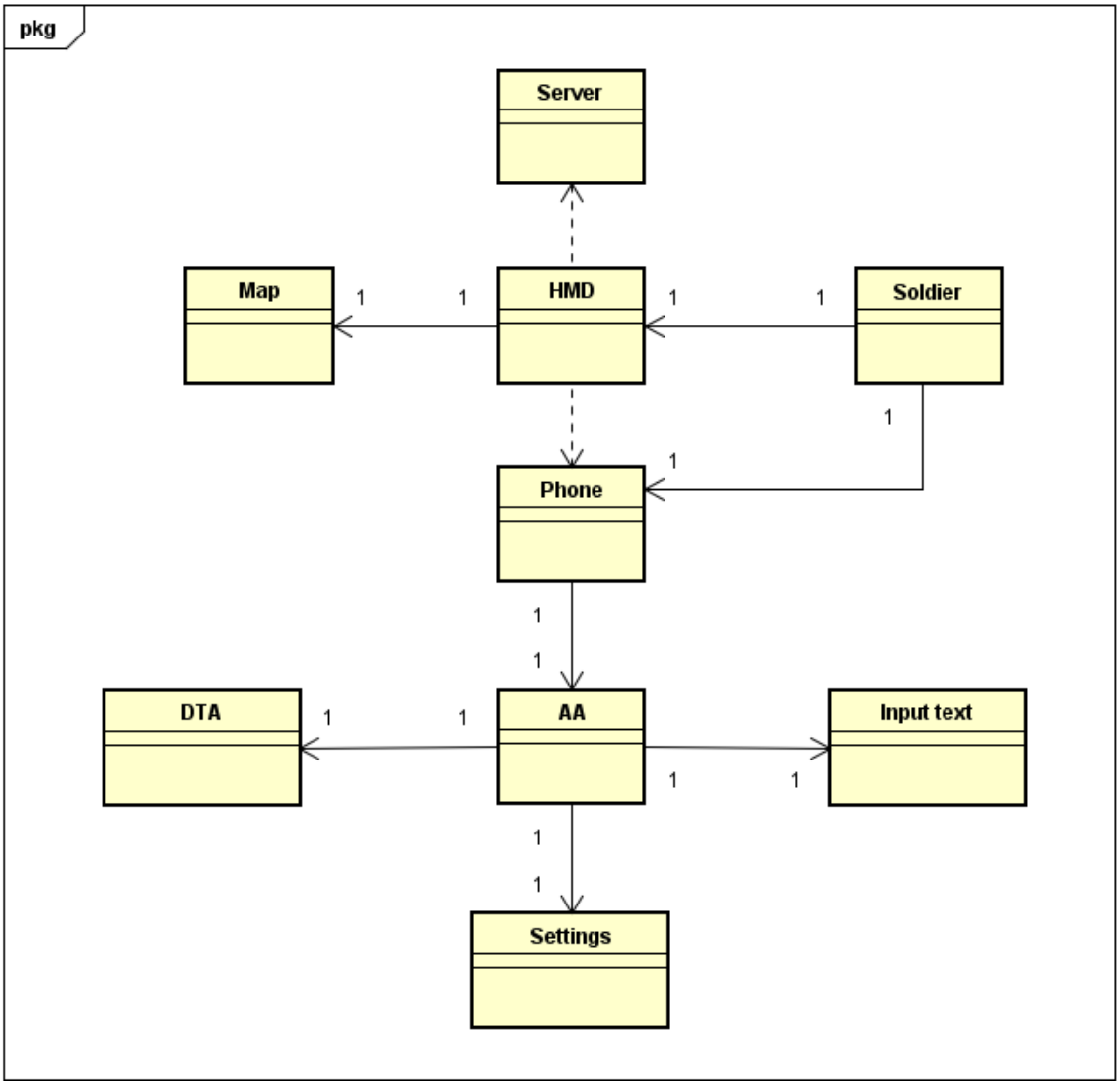


Figure 4 - Domain model

5 Design

This section outlines how the system is structured from a development point of view. Descriptions of the architecture of the system and the technologies involved can be found here, along with a class diagram, interaction diagrams and the choice of design patterns.

5.1 Architecture

The team has considered using several different software architectures for the project, such as MVC (model-view-controller), MVVM (model-view-viewmodel) and MVP (model-view-presenter).

MVC provides a good solution by not having the business logic in the UI (activities in the project's case) and it's easy to perform unit tests on. However, the controller might grow too big, since it would be the middleman between all activities and the model and violate some of the SOLID principles (Single responsibility and Interface segregation) because of making some activities dependable on methods not usable by said activities.

The next considered architecture was MVVM, which had a lot of positive aspects, including but not limited to:

- Very easily testable
- Compile time check for database entities
- Presentation layer is in XML
- Backed-up by Google using Android JetPack
- Data binding between the View and the ViewModel

However, data binding might not prove to be a solution for all types of systems and violates the Single responsibility principle of SOLID.

MVP is based on the same principle as MVC, with the Presenter acting as the middleman between the View and the Model. From the point of view of the AA, the View would consist of the Activities' XML design, the Presenter would be the Activity itself and the Model would consist of the persistent data represented by java classes. It is also easy to test and helps splitting complex tasks into more simpler ones. (MVC, MVP, MVVM Design Patterns with Godfrey Nolan - YouTube, 2018)

However, due to the simplicity of the project and the fact, that the system is not required to save data in a persistent manner the use of a software architecture seemed unnecessary.

Therefore, the team has decided to implement the code in a modular way, thus not using any known system architectures, but still trying to respect the SOLID principles.

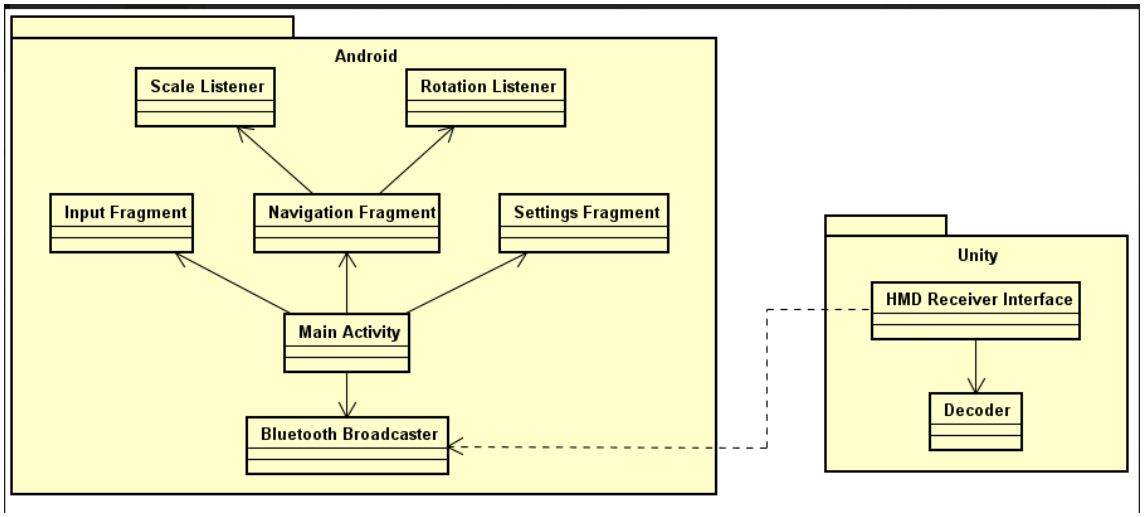


Figure 5 - Simplifed Use Case Diagram

5.2 Technologies

5.2.1 HMD

The current SitaWare MR implementation is based on Microsoft HoloLens, hence this was the mixed reality HMD of choice for the project. "Microsoft HoloLens is the first

self-contained, holographic computer, enabling you to engage with your digital content and interact with holograms in the world around you." (Microsoft HoloLens | The leader in mixed reality technology, 2018)

5.2.2 AA

Android is a reliable technology, open source, has an easily customizable user interface, and it is widely used all around the world. For these reasons the usage of an Android device was the best choice for a remote controller prototype for an HMD.

"The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model. Unlike programming paradigms in which apps are launched with a main() method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle." (Introduction to Activities | Android Developers, 2018) Stages that can be observed in Figure 6.

"An activity provides the window in which the app draws its UI. This window typically fills the screen but may be smaller than the screen and float on top of other windows. Generally, one activity implements one screen in an app. Most apps contain multiple screens, which means they comprise multiple activities. Typically, one activity in an app is specified as the *main activity*, which is the first screen to appear when the user launches the app." (Introduction to Activities | Android Developers, 2018)

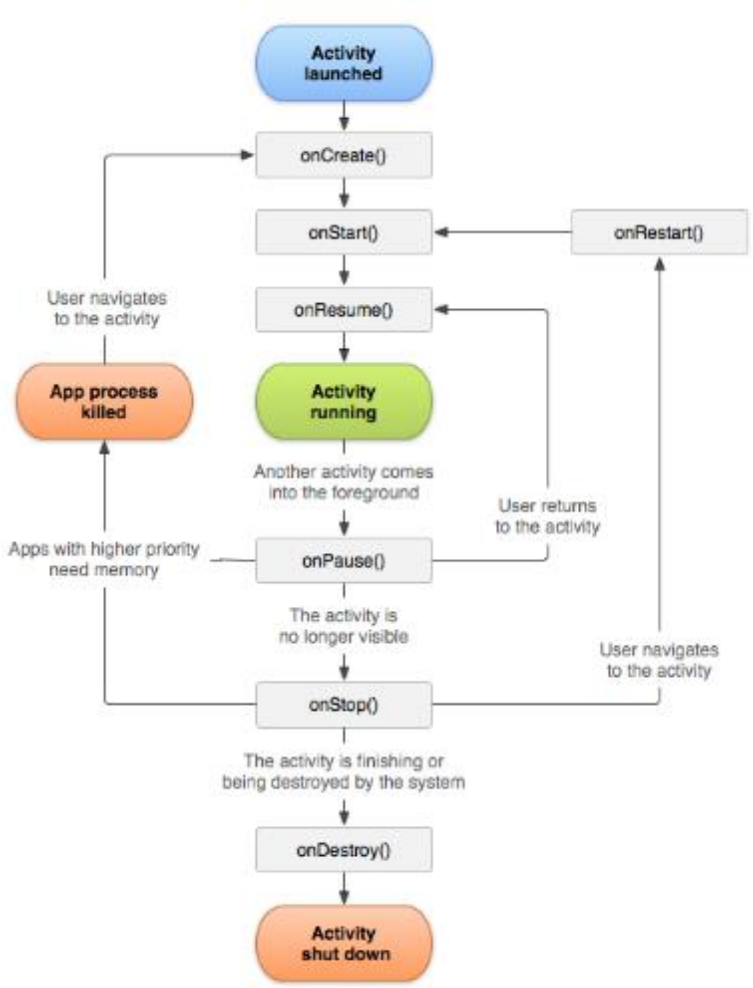


Figure 6 - Android Activity lifecycle(Introduction to Activities | Android Developers, 2018)

The application only has one activity, which is always running. The activity consists of 3 fragments, one for each view (Input, Navigation, Settings, explained in detail below). Only one fragment is displayed at a time, and a menu bar at the bottom makes it possible for the user to switch between the fragments, while the lifecycle state of the activity remains unchanged.

"A Fragment represents a behavior or a portion of user interface in a FragmentActivity. You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities. You can think of a fragment as a modular section

of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running (sort of like a "sub activity" that you can reuse in different activities).

A fragment must always be hosted in an activity and the fragment's lifecycle is directly affected by the host activity's lifecycle." (Fragments | Android Developers, 2018) The fragment's lifecycle can be observed in Figure 7.

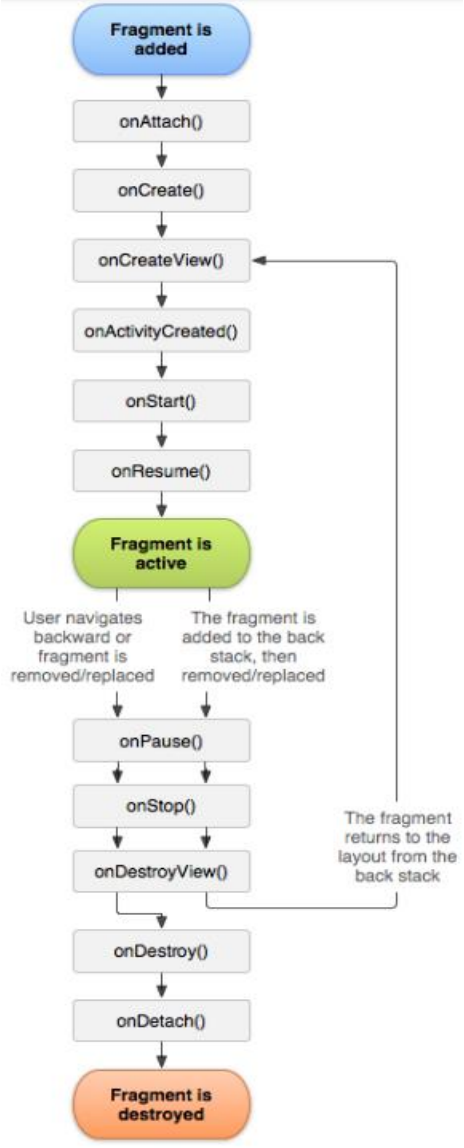


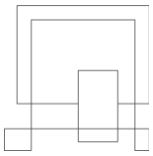
Figure 7 - Fragment lifecycle(Fragments | Android Developers, 2018)

The main reason for this design is the aspect of resource management, as this implementation ensures, that there is always going to be only one living activity while using the application (keeping memory usage at an optimal level).

Out of the mentioned lifecycle methods, the ones used are:

“Activity

- onCreate - fires when the system first creates the activity. On activity creation, the activity enters the *Created* state. In the onCreate() method, basic application startup logic is performed that should happen only once for the entire life of the activity.
- onResume - when entered, the activity comes to the foreground, and then the system invokes the onResume()callback. This is the state in which the app interacts with the user. The app stays in this state until something happens to take focus away from the app. Such an event might be, for instance, receiving a phone call, the user’s navigating to another activity, or the device screen turning off.
- onPause - The system calls this method as the first indication that the user is leaving the activity (though it does not always mean the activity is being destroyed); it indicates that the activity is no longer in the foreground (though it may still be visible if the user is in multi-window mode). The method is used to pause or adjust operations that should not continue (or should continue in moderation) while the Activity is in the Paused state. Typically used to release system resources, handles to sensors (like GPS), or any resources that may affect battery life while your the activity is paused.
- onStop - When the activity is no longer visible to the user, it has entered the *Stopped* state, and the system invokes the onStop() callback. This may occur, for example, when a newly launched activity covers the entire screen. The system



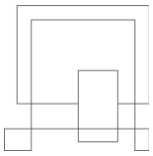
may also call onStop() when the activity has finished running, and is about to be terminated." (Introduction to Activities | Android Developers, 2018)

Fragment

- onCreateView – "Called to have the fragment instantiate its user interface view. To draw a UI for your fragment, you must return a View from this method that is the root of your fragment's layout. You can return null if the fragment does not provide a UI." (Fragments | Android Developers, 2018)
- onResume is the same but using a Fragment instead of an Activity.

5.2.3 Connection between the HMD and the AA

As far as the establishment of the connection between the HMD and the AA was concerned the team has considered the usage of one of three different technologies. The three technologies in discussion were Bluetooth, Bluetooth Low Energy and Rest API.
The idea to use Rest API was abandoned due to the fact, that it has the highest latency among these technologies. This might become an issue as far as the effectiveness of the AA is concerned.
Bluetooth seemed to be the most viable option, however, after further investigation the team had found out, that it is not possible to create a direct connection between HoloLens and an Android device.
Therefore, the technology chosen for the implementation of the project is Bluetooth Low Energy, because it is low latency, it provides a local connection between the devices, saves battery lifetime and it allows data transfer to a sufficient extent for the project's needs.
BLE works using GATT (General ATTribute profile). GATT describes how to transfer small amounts of data via low-power **radio** waves. The profile consists of services, and those services consists of characteristics.
A more general way to understand what a service is in GATT, is to compare it with a class in most object-oriented languages, complete with instantiation, because a service



can be instantiated multiple times within a single GATT server (however, this is not a common occurrence and most services would therefore be akin to singletons). (4. GATT (Services and Characteristics) - Getting Started with Bluetooth Low Energy [Book], 2018)

Characteristics represent the actual user data broadcasted by the BLE, split in two attributes. The first attribute holds the metadata that provides some information about the actual user data which is found in the second attribute. The characteristic can also contain some descriptors that which provides more insight in the metadata provided in the characteristic declaration.

A BLE device may operate in three different modes depending on required functionality: advertising, scanning and initiating. In the advertising mode, the device periodically sends data via BLE by broadcasting on 3 channels (37, 38, 39). In the meantime, the receiver actively scans the 3 channels for a predetermined time frame. (Cho et al., 2014)

For this implementation, the Android Device acts as the BLE server, broadcasting an array of bytes to the HMD (which acts as the BLE client), containing instructions to be decoded in the HMD, resulting in the user interaction with the SitaWare solution.

The way the data is sent from the AA to the HMD was designed in a way, that it only sends one data packet at a time, instead of continuously sending data packets in real time. The reason for this is that in the early implementations of the AA it was discovered, that sending too many data packets in a fast, consequential order over a BLE connection, it can cause the BLE broadcaster to stop working. In the case of the AA, upon sending too many data packets in a small time frame the AA stopped broadcasting the BLE signal, and it could only be started again by rebooting the device itself.

This problem was fixed by making the broadcaster send only one data packet at a time, which was achieved by making certain changes in the code (explained in more details in the *Implementation* section).

In order for the HMD interface to understand the instructions the AA is sending; a custom protocol was established. Because of BLE packets being able to contain 31 (Bluetooth Low Energy Scanning and Advertising, 2018) bytes of custom advertisement data, the following rules were set:

- First byte of the custom data would contain the instruction category (e.g. panning the map, rotating the map, etc.)
- The second byte would contain the actual instruction the HMD needs to perform on the holographic map (except the user text input).
- For the user text input, the remainder 30 bytes are used to send a text containing a maximum of 30 characters.

Table 1 - Bluetooth Protocol Table

Instruction category	Instruction category ID	Instruction type	Instruction type ID
Panning map	1	North	1
		South	2
		East	3
		West	4
Zooming map	2	Zoom in	1
		Zoom out	2
Rotating map	3	Counter clockwise rotation	1
		Clockwise rotation	2
Text input	4	Custom text input	Custom value entered by user
Scaling map	5	Scale down	1
		Scale up	2
Selection tap	6	Selection tap	1

5.2.4 Unity

The Unity game engine provides a useful framework for creating applications for the HoloLens, making the development process quick and simple. Moreover, Systematic's current implementation of the HoloLens application was made with Unity. Thus, Unity became the software of choice for the team for developing a mock-up application for the HoloLens.

5.3 Design Patterns

In terms of design patterns, the following patterns were implemented:

- Singleton was used for the Broadcaster class which manages the BLE advertisement, because the class uses the Bluetooth Adapter and, in this way, instantiating multiple broadcasters would be avoided, since only one class that advertises in a continuous way is needed.

5.4 Class Diagram

Please refer to Appendix C – Class & sequence diagrams

5.5 Sequence Diagram

The following sequence diagram (Figure 8) describes in more detail the chain of events happening upon user interaction, and the different entities partaking in the process. The example below is the sequence diagram of the Navigation screen (the sequence diagram

for the Input and the Settings screen can be found in Appendix C – Class & sequence diagrams).

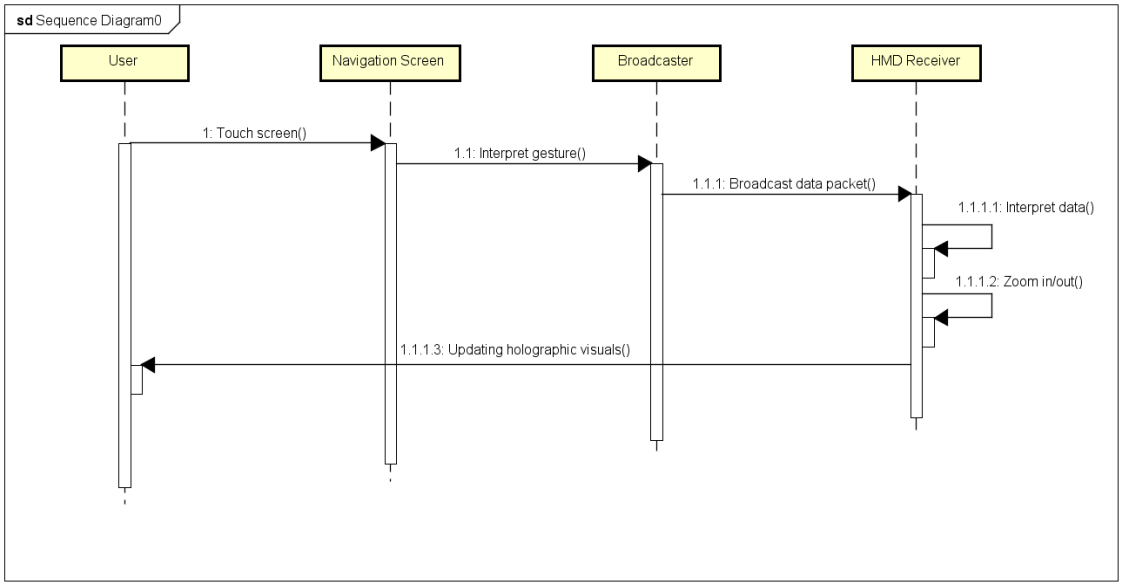


Figure 8 - Sequence Diagram

First, the *user* performs a touch gesture on the *Navigation screen* of the AA. The touch gesture is interpreted by the application, determining what kind of touch gesture did the user perform (e.g. pinching, tapping, etc.). Based on the interpreted gesture the *broadcaster* creates a data packet (which defines a command for the HMD) and sends it to the *HMD receiver*. The receiver processes the data packet and performs the required action (e.g. zooming in/out upon a pinching gesture), thus updating the holographic visuals the user sees.

5.6 Navigation input

Different types of gestures need to be supported in order to interact with the holographic map. To differentiate between the more complex ones, which require 2 fingers, the AA needs clear boundaries to make sure the correct gesture was made. The following activity diagram (Figure 9) describes the logic behind gesture recognition and

distinction. *(For the whole diagram please refer to Appendix B – Use case & activity diagrams)*

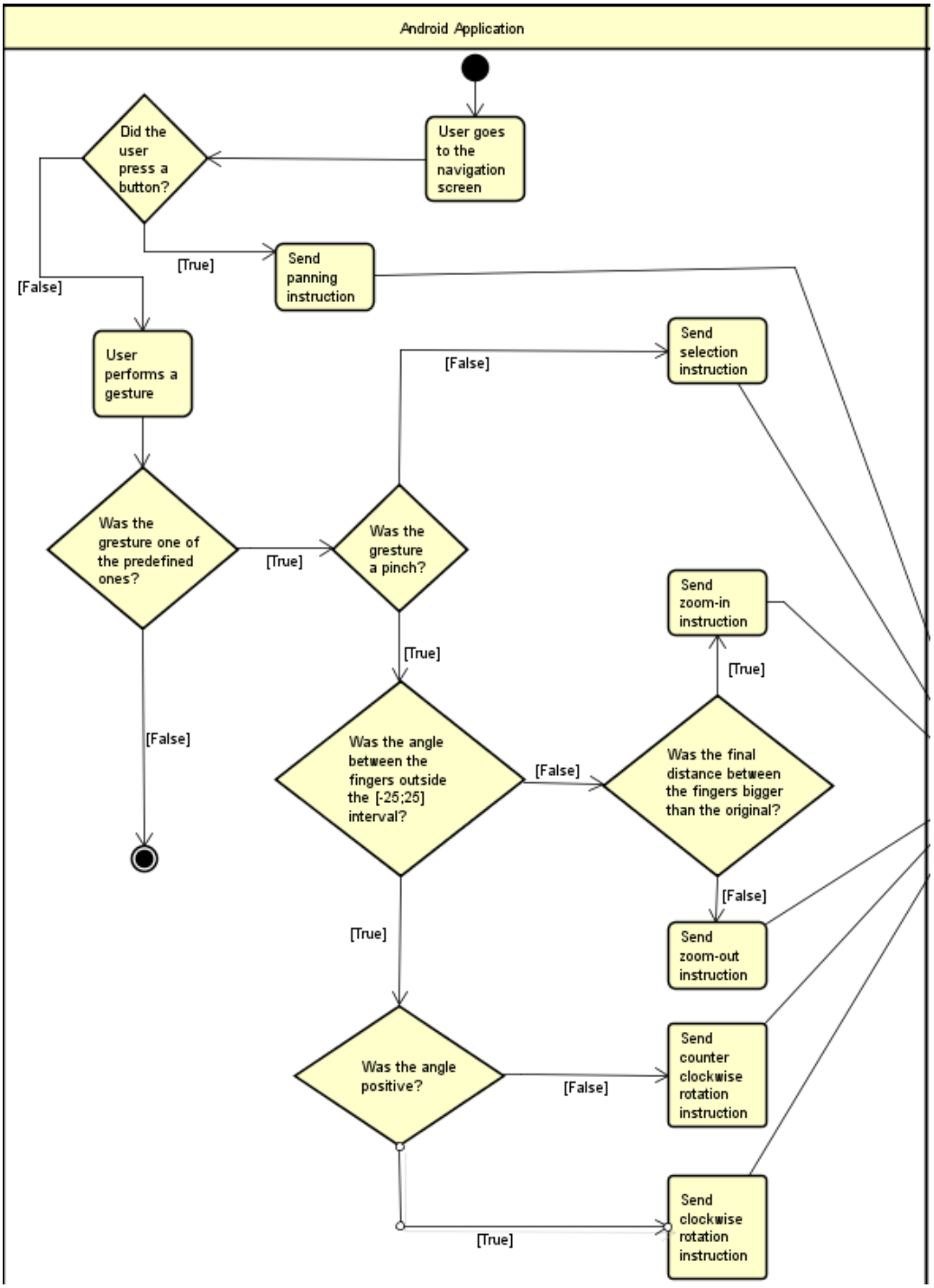


Figure 9 - Activity Diagram describing gesture recognition flow

5.7 UI design choices

During the interview with the stakeholder, the team was provided with some conceptual ideas and feedback regarding how the user interface of the AA should look like. The domain expert gave some insight concerning the way the end-user would likely interact with the AA. Thus, the following drafts are envisioning a UI suitable for the end-user's needs, requiring minimum training.

The application consists of three views: **Input**, **Navigation** and **Settings**. Switching between the views is possible using a menu bar located on the bottom of the screen in portrait orientation, and on the right side of the screen in landscape orientation. The menu bar consists of three buttons, I, N and S, each corresponding to the aforementioned activities respectively.

The **Navigation** view is the default view showed when the application launches, since it contains the most important functionalities of the application. In this view the user can find a DTA, and four arrow keys on each side of it. On the DTA the user can perform Android's built in touch gestures: two finger pinching gesture for zooming in the map, two finger rotation gesture for rotating the map and tapping for selection on the map. The buttons can be used for panning the map. Figures 10 and 11 show how the view would look like.

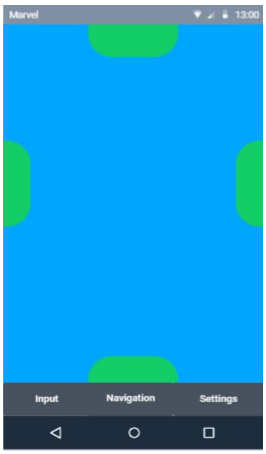


Figure 10 - Navigation view prototype

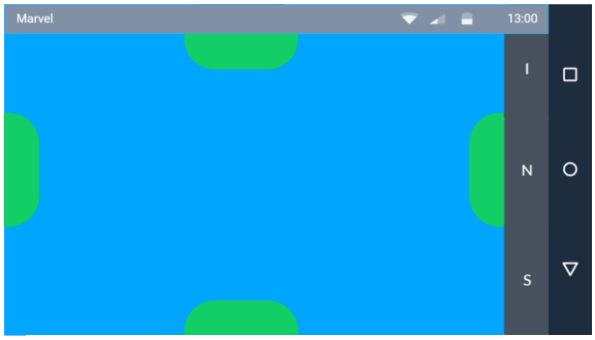


Figure 11 - Navigation view prototype (landscape)

The **Input** view would display a text field and a submit button. In the lower part of the screen, the default Android keyboard would be displayed whenever the user interacts with the above-mentioned text field. When pressing the submit button, the text entered by the user would be transmitted to the HMD. Figures 12 and 13 show how the view would look like.

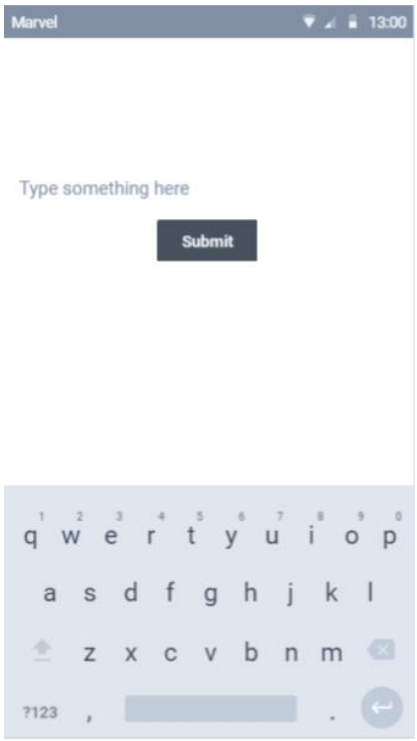


Figure 12 - Input view prototype

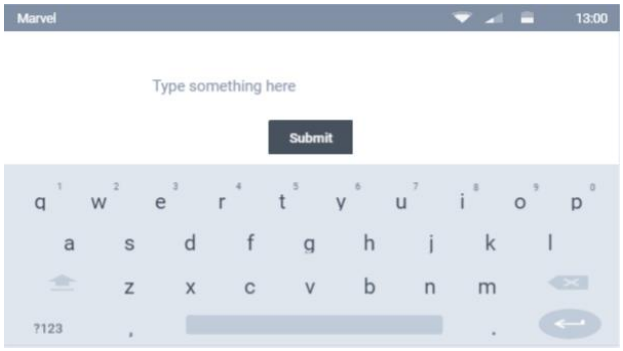


Figure 13 - Input view prototype (landscape)

In the **Settings** view, the user can find a scale slider, a checkbox for locking the application's orientation (portrait or landscape) and a Bluetooth status message.

With the scale slider the user can adjust the size of the map to their preferences in accordance with the predefined valid boundaries.

The Bluetooth status message is a label, which would inform the user whether the AA is broadcasting a Bluetooth Low Energy signal or not. Figures 14 and 15 show how the activity would look like.

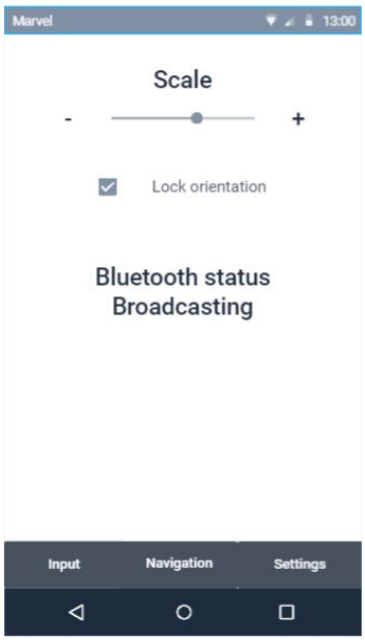


Figure 14 - Settings view prototype

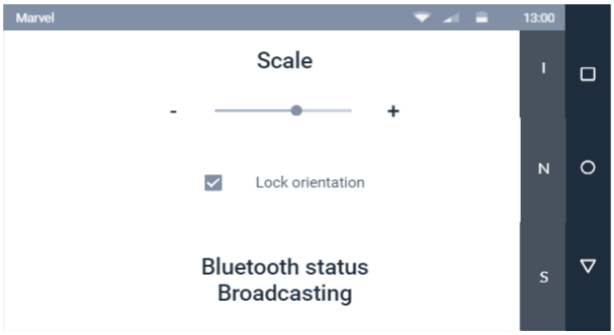


Figure 15 - Settings view prototype (landscape)

6 Implementation

This section describes the development of the system, including, but not limited to implementation of design patterns and a walkthrough from the UI to the "back-end" of the system.

6.1 The AA

In this section interesting code snippets and explanations about the implementation can be found, regarding classes on the **AA** side. This includes the **BLE Broadcaster**, the **Main Activity** and the 3 fragments of the application, **Navigation, Input** and **Settings**. As mentioned before, all **AA** related code was written in **Java (**importing the **Android libraries)** with the use of **Android Studio**.

6.1.1 BLE Broadcaster

As mentioned before, the application is capable of broadcasting Bluetooth Low Energy signals. The **Broadcaster** class is responsible for that functionality.

The class itself implements the Singleton design pattern (Figure 16), in order to make sure there's only going to be one instance of the **Broadcaster** (since we only need one **Broadcaster** object at a time).

```
public static Broadcaster getInstance(BluetoothManager manager, BluetoothAdapter mBAdapter,
                                       BluetoothLeAdvertiser mBLEAdvertiser, Handler mainHandler)
{

    if(INSTANCE == null){
        INSTANCE = new Broadcaster(manager, mBAdapter, mBLEAdvertiser, mainHandler);
    }
    return INSTANCE;

}
```

Figure 16 - Code snippet of the singleton class Broadcaster

```
private void startAdvertising() {
    if (mBLEAdvertiser == null) return;
    AdvertiseSettings settings = new AdvertiseSettings.Builder()
            .setAdvertiseMode(AdvertiseSettings.ADVERTISE_MODE_LOW_LATENCY)
            .setConnectable(false)
            .setTimeout(800)
            .setTxPowerLevel(AdvertiseSettings.ADVERTISE_TX_POWER_ULTRA_LOW)
            .build();
    mBLEAdvertiser.startAdvertising(settings, data, mAdvertiseCallback);
    try {
        sleep( millis: 200);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public void stopAdvertising() {
    if (mBLEAdvertiser == null) return;
    mBLEAdvertiser.stopAdvertising(mAdvertiseCallback);
    mainHandler.sendMessage(Message.obtain( h: null,  what: 0, new Message()));
}

private void restartAdvertising() {
    stopAdvertising();
    startAdvertising();
}
```

Figure 17 - Code snippet of Bluetooth broadcaster methods and settings

The data about the actions to be performed are saved in data packets, which are sent to the HoloLens receiver by the **Broadcaster**. Advertisement of the data packets is started with the **startAdvertising** method. In that method the advertiser object (*mBLEAdvertiser*) of the class is used to start advertising the data packets, calling Android's built in **startAdvertising** method (from the class *BluetoothLeAdvertiser*), which takes 3 parameters, *settings*, *data* and *mAdvertiseCallback*.

Data is the data packet prepared for sending. The creation of the data packets is explained in more detail below.

The *mAdvertiseCallback* object is used to send notifications whether the advertisement of the data is running or not. It also restarts the advertisement in case of a failure.

The *settings* for the advertisement are also prepared in this method. *(as seen in Figure 17)*

First of all, the advertisement mode is set to low latency, since the application requires low latency for acceptable user experience (.*setAdvertisementMode*).

Secondly, the advertisement is set to non-connectable, since HoloLens doesn't support direct connections with Android devices(.*setConnectable*).

Thirdly, time out of the advertisement is also set to 800 milliseconds (.*setTimeout*). The value of 800 was chosen based on trial and error. The application is more prone to defects on different values: below 800 ms, the sent data might not be received by the HoloLens receiver, and above 800 ms the same data might be sent twice instead of only one time (discovered during implementation by black box testing).

Finally, the TX power level of the advertisement is kept on the ultra-low setting, since the distance between the AA and the HMD is supposed to be very short, thus higher power levels are unnecessary (.*setTxPowerLevel*).

```
private byte[] buildBLEPacket(byte id, byte[] payload) {
    byte[] packet = new byte[payload.length + 1];
    packet[0] = id;
    System.arraycopy(payload, i: 0, packet, i1: 1, payload.length);
    Log.d( tag: "Packet to be sent", Arrays.toString(packet));
    return packet;
}

// IDs:
// 1 - Panning
// 2 - Zooming
// 3 - Rotating
// 4 - Text input
// 5 - Scaling
// 6 - Tap
public void createPacketWithData(byte id, byte[] payload) {
    data = new AdvertiseData.Builder()
            .addManufacturerData(BEACON_ID, buildBLEPacket(id, payload))
            .build();
    startAdvertising();
}
```

Figure 18 - Code snippet of the method that is responsible of building the BLE packets

Two methods handle the building of the data packets: the **buildBLEPacket** and the **createPacketWithData** methods. *(as seen in Figure 18)*

The **createPacketWithData** method takes 2 arguments, an *ID* (of type byte) and a *payload* (an array of bytes). *ID* corresponds to the type of action that should be executed (e.g. *ID* = 1 for panning, *ID* = 2 for rotation, etc.). *Payload* contains a byte array which designates a specific command inside the different functionalities. For example, in case the user performs a "pinch-in" gesture (i.e. zooming in, *ID* = 2) *payload* will be equal to 1, and in case of a "pinch-out" gesture it will be equal to 2. *Payload* will only contain one byte for each functionality, with the exception of the input functionality (it will contain a byte for each character in the string the user wants to send).

For the actual creation of the data packet the **createPacketWithData** method loads an *AdvertiseData* object, "*data*", with the *ID* and the *payload* of the received gesture using the **buildBLEPacket** method.

The **buildBLEPacket** method creates and returns a packet object, which is an array of bytes. Then it stores the *ID* and the *payload* in it (*ID* will be the first element of the array, followed by the byte(s) in the *payload*).

6.1.2 Main activity

```
public void navBarHandler() {
    setContentView(R.layout.activity_main);
    navigation = findViewById(R.id.navigation);
    navigation.setOnNavigationItemSelectedListener(new BottomNavigationView.OnNavigationItemSelectedListener() {
        @Override
        public boolean onNavigationItemSelected(@NonNull MenuItem item) {
            Fragment selectedFragment = null;
            switch (item.getItemId()) {
                case R.id.navigation_navigation:
                    selectedFragment = new NavigationFragment();
                    editor.putString( s: "selectedFragment",  s1: "Navigation");
                    editor.commit();
                    break;

                case R.id.navigation_settings:
                    selectedFragment = new SettingsFragment();
                    editor.putString( s: "selectedFragment",  s1: "Settings");
                    editor.commit();
                    break;

                case R.id.navigation_input:
                    selectedFragment = new InputFragment();
                    editor.putString( s: "selectedFragment",  s1: "Input");
                    editor.commit();
                    break;
            }
            getSupportFragmentManager().beginTransaction().replace(R.id.frame, selectedFragment).commit();

            return true;
        }
    });
}
```

Figure 19 - Code snippet of the logic behind the menu bar and the fragments

Method **navBarHandler** in **MainActivity** handles the navigation bar in the bottom of the activity. It sets a listener on the bar, which waits for action, and depending on the selection it displays the corresponding fragments inside of the activity in the application. It also sets the **selectedFragment** key in the shared preferences, which is used to store which fragment is currently being in use, so that it would persist throughout the app's lifecycle. *(as seen in Figure 19)*

```
private void orientationFragmentHandler() {
    if (pref.getString( s: "selectedFragment",  s1: "defaultValue").equals("Navigation")
            || pref.getString( s: "selectedFragment",  s1: "defaultValue").equals("defaultValue")) {
        getSupportFragmentManager().beginTransaction().replace(R.id.frame, new NavigationFragment()).commit();
        navigation.getMenu().findItem(R.id.navigation_navigation).setChecked(true);

        Menu menu = navigation.getMenu();
        MenuItem menuItem = menu.getItem( i: 1);
        menuItem.setChecked(true);
    } else if (pref.getString( s: "selectedFragment",  s1: "defaultValue").equals("Settings")) {
        getSupportFragmentManager().beginTransaction().replace(R.id.frame, new SettingsFragment()).commit();
        navigation.getMenu().findItem(R.id.navigation_settings).setChecked(true);

        Menu menu = navigation.getMenu();
        MenuItem menuItem = menu.getItem( i: 2);
        menuItem.setChecked(true);
    } else {
        getSupportFragmentManager().beginTransaction().replace(R.id.frame, new InputFragment()).commit();
        navigation.getMenu().findItem(R.id.navigation_input).setChecked(true);

        Menu menu = navigation.getMenu();
        MenuItem menuItem = menu.getItem( i: 0);
        menuItem.setChecked(true);
    }
}
```

Figure 20 - Code snippet of the logic behind showing the correct fragment on orientation change

orientationFragmentHandler (Figure 20) gets called on app opening and on orientation change of the phone. It checks what the **selectedFragment** key contains and depending on that it shows a different fragment and sets the selected one in the navigation UI.

```
<activity android:name=".MainActivity"
    android:configChanges="orientation">
```

Figure 21 - Code snippet of the Android Manifest file

```
// executed on orientation change
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);
    navBarHandler();
    orientationFragmentHandler();
    bluetoothHandler();
}
```

Figure 22 - Code snippet of the method called on orientation change

An interesting part of the **MainActivity** is the way orientation is handled. When a configuration change occurs at runtime, the activity is shut down and restarted by default, but declaring a configuration with this attribute (Figure 21) will prevent the activity from being restarted. Instead, the activity remains running and its **onConfigurationChanged** method is called (Figure 22). This is used in order to get around the call of **onCreate** on rotation change.

The Navigation fragment is the main and starting point of the AA. In order to achieve that, the onStop method is overridden so that the *selectedFragment* **sharedPref** is set to contain navigation (Figure 23). The next time the AA is opened **orientationFragmentHandler** call in **onCreate** would replace the frame with the navigation fragment.

```
// executed only on startup(Not on orientation change)
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    pref = getApplicationContext().getSharedPreferences( s: "PrefNavBar",  i: 0); // 0 - for private mode
    editor = pref.edit();
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    navBarHandler();
    orientationFragmentHandler();
    bluetoothHandler();
}

@Override
protected void onStop() {
    editor.putString( s: "selectedFragment",  s1: "Navigation");
    editor.commit();
    super.onStop();
```

Figure 23 - Code snippet of the Overridden Android methods

bluetoothHandler (Figure 24) instantiates the Bluetooth manager and adapter and creates a new instance of the **Singleton** class **Broadcaster.**

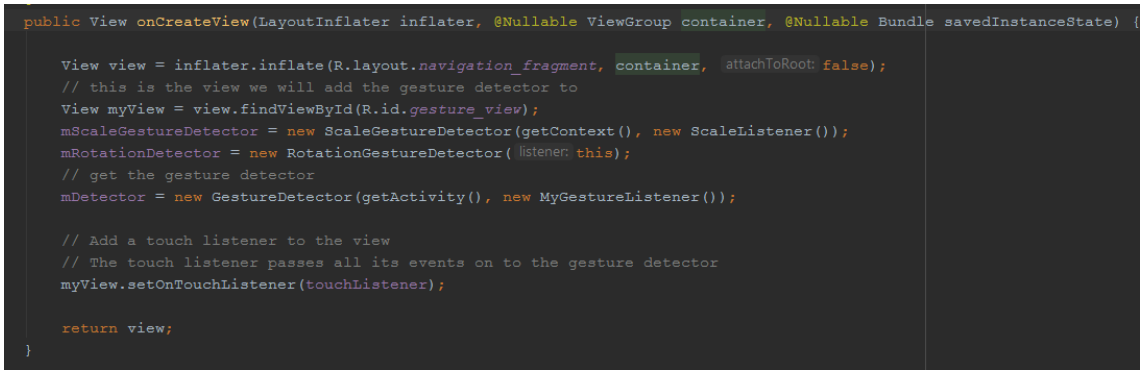
```
private void bluetoothHandler() {
    mainThreadHandler = (Handler) handleMessage(msg) → {
            if (msg.what == 1) {
                status = "Broadcasting";
                Log.d( tag: "BLEStatus", status);
            } else if (msg.what == 0) {
                status = "Not Broadcasting";
                Log.d( tag: "BLEStatus", status);
            }
    };

    mBManager = (BluetoothManager) getSystemService(BLUETOOTH_SERVICE);
    if (mBManager != null) {
        mBAdapter = mBManager.getAdapter();
        if (mBAdapter != null) {
            mBLEAdvertiser = mBAdapter.getBluetoothLeAdvertiser();
            broadcaster = Broadcaster.getInstance(mBManager, mBAdapter, mBLEAdvertiser, mainThreadHandler);
        }
    }
}

private void orientationLock() {
    SharedPreferences prefCheckbox = getApplicationContext().getSharedPreferences( s: "CheckBoxPref",  i: 0);
    if (prefCheckbox.getBoolean( s: "checkbox",  b: false) == true) {
        if (getResources().getConfiguration().orientation == Configuration.ORIENTATION_PORTRAIT) {
            setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
        } else {
            setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
        }
    } else setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_UNSPECIFIED);
}
```

Figure 24 - Code snippet of the logic behind the instantiation of the Bluetooth part

orientationLock gets the **SharedPreference's checkbox** key created in **SettingsFragment** and depending on whether it contains true or false it locks or unlocks the orientation. **orientationLock** is being called in onResume, since the lock of orientation needs to persist once the AA is closed or moved to the background.

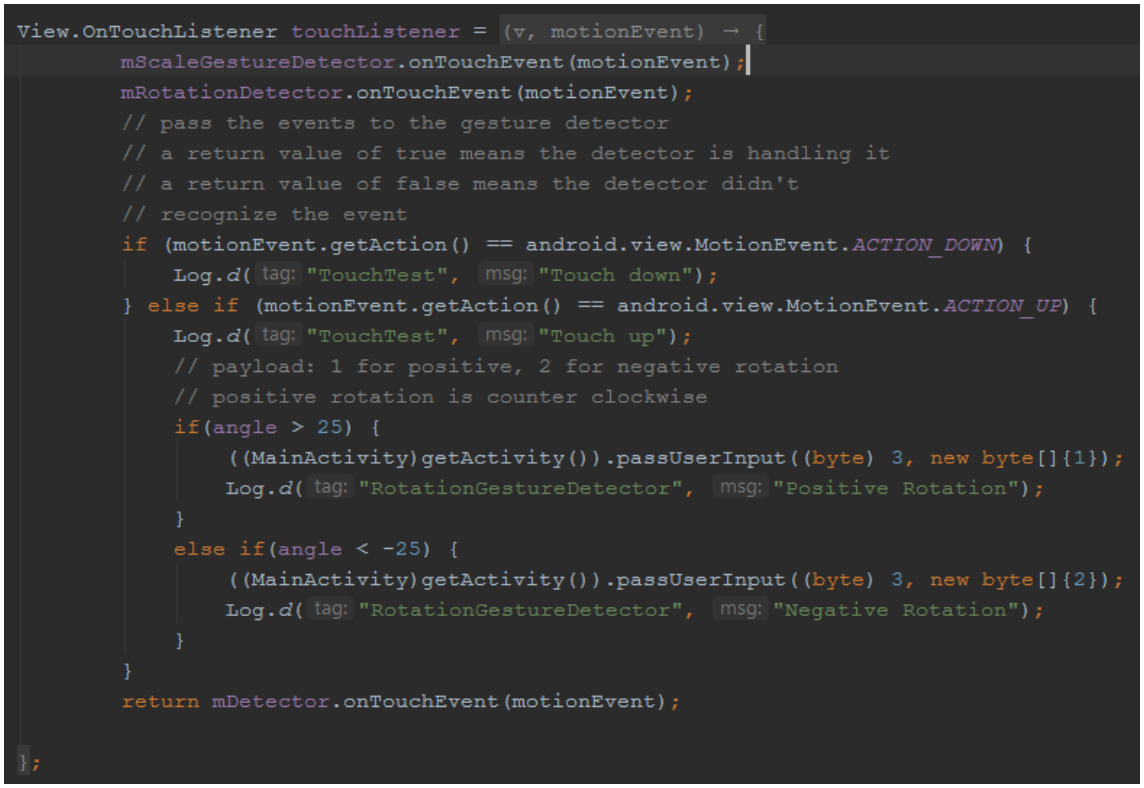
6.1.3 Navigation fragment

NavigationFragment is the main fragment of the application which contains all the navigation functionalities` logic.



```
public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {

    View view = inflater.inflate(R.layout.navigation_fragment, container, attachToRoot: false);
    // this is the view we will add the gesture detector to
    View myView = view.findViewById(R.id.gesture_view);
    mScaleGestureDetector = new ScaleGestureDetector(getContext(), new ScaleListener());
    mRotationDetector = new RotationGestureDetector( listener: this);
    // get the gesture detector
    mDetector = new GestureDetector(getActivity(), new MyGestureListener());

    // Add a touch listener to the view
    // The touch listener passes all its events on to the gesture detector
    myView.setOnTouchListener(touchListener);

    return view;
}
```

Figure 25 - Code snippet of how a fragment is instantiated



```
View.OnTouchListener touchListener = (v, motionEvent) -> {
        mScaleGestureDetector.onTouchEvent(motionEvent);
        mRotationDetector.onTouchEvent(motionEvent);
        // pass the events to the gesture detector
        // a return value of true means the detector is handling it
        // a return value of false means the detector didn't
        // recognize the event
        if (motionEvent.getAction() == android.view.MotionEvent.ACTION_DOWN) {
            Log.d( tag: "TouchTest", msg: "Touch down");
        } else if (motionEvent.getAction() == android.view.MotionEvent.ACTION_UP) {
            Log.d( tag: "TouchTest", msg: "Touch up");
            // payload: 1 for positive, 2 for negative rotation
            // positive rotation is counter clockwise
            if(angle > 25) {
                ((MainActivity)getActivity()).passUserInput((byte) 3, new byte[]{1});
                Log.d( tag: "RotationGestureDetector", msg: "Positive Rotation");
            }
            else if(angle < -25) {
                ((MainActivity)getActivity()).passUserInput((byte) 3, new byte[]{2});
                Log.d( tag: "RotationGestureDetector", msg: "Negative Rotation");
            }
        }
        return mDetector.onTouchEvent(motionEvent);

};
```

Figure 26 - Code snippet of the logic behind the gesture listeners

onCreateView (Figure 25) inflates the navigation fragment in the **MainActivity**, instantiates the scaling, rotating and gesture detectors with the corresponding listeners and adds a touch listener to the view. In the **touchListener's onTouch** (Figure 26) the events are passed to the gesture detectors. Touch gestures are handled by three different gesture listeners:

- Gesture detector, which handles the basic Android gestures (e.g. tapping, scrolling, flinging etc.)
- Scale gesture detector, which handles the pinching gesture by comparing the span of the user's fingers movement to the default value of 128 in order to differentiate between a pinch-in and a pinch-out gesture
- Rotation gesture detector, which handles the rotation gesture by calculating the angle the user's fingers have done from when the interaction starts until when the user takes its fingers off the screen

Rotation is handled in a way that on finger release a check of angle is done and depending whether it's more than 25 or less than -25 a user input consisting of positive or negative rotation is sent to the broadcaster which creates the packet and sends it via BLE. The previous mentioned check is done in order to differentiate when the user performs a zoom or a rotation gesture.

```
private class ScaleListener extends ScaleGestureDetector.SimpleOnScaleGestureListener {

    private float originalValue = 128;

    @Override
    public boolean onScale(ScaleGestureDetector scaleGestureDetector){
        mScaleFactor *= scaleGestureDetector.getScaleFactor();
        mScaleFactor = Math.max(1.0f,
                Math.min(mScaleFactor, 256.0f));
        Log.d( tag: "TAG",  msg: "Scale factor: " + Float.toString(mScaleFactor));
        return true;
    }

    @Override
    public boolean onScaleBegin(ScaleGestureDetector detector){
        Log.d( tag: "Scale",  msg: "It started");
        mScaleFactor = originalValue;
        return true;
    }

    @Override
    public void onScaleEnd (ScaleGestureDetector detector){

        if(mScaleFactor > originalValue && Math.abs(angle) < 25) {
            // 1 for zoom in
            ((MainActivity)getActivity()).passUserInput((byte) 2, new byte[]{1});
            Log.d( tag: "Scale",  msg: "Scale factor: " + Float.toString(mScaleFactor));
            Log.d( tag: "Scale",  msg: "Zoomed in");
        } else if (mScaleFactor < originalValue && Math.abs(angle) < 25){
            // 0 for zoom out
            ((MainActivity)getActivity()).passUserInput((byte) 2, new byte[]{2});
            Log.d( tag: "Scale",  msg: "Scale factor: " + Float.toString(mScaleFactor));
            Log.d( tag: "Scale",  msg: "Zoomed out");
        }
        Log.d( tag: "Scale",  msg: "It ended");
    }
}
```

Figure 27 - Code snippet of the logic behind the scale gesture detector

On the other hand, scaling is handled by firstly resetting the **scaleFactor** to its default value and then calculating the scale factor in the o**nScale** interaction event. Finally, **onScaleEnd** is fired and user input consisting of zoom-in/zoom-out is sent to the broadcaster which creates the packet and sends it via BLE. *(as seen in Figure 27)*

6.1.4 Input fragment

The **InputFragment** is the fragment being used to handle user input in the AA which contains one single text field and a button to send. The text's maximum size is 30 bytes

because the custom data capacity of the BLE's packet is 31 bytes, with 1 byte being reserved for the category of the instruction being sent. On button click the text is converted to bytes and sent to the broadcaster which creates the packet and sends it via BLE. The HMD that receives the text through BLE has to convert back the text from bytes to string.

6.1.5 Settings fragment

SettingsFragment contains a lock orientation checkbox feature (Figure 28), map size buttons and a Bluetooth status message displayed.

```
@Override
public void onClick(View view) {
    switch (view.getId()) {
        case R.id.minus:
            changeMapSize((byte) 1);
            break;
        case R.id.plus:
            changeMapSize((byte) 2);
            break;
        case R.id.checkBox:
            if (checkBox.isChecked()) {
                editor.putBoolean( s: "checkbox",  b: true);
                editor.commit(); // commit changes
                if (getResources().getConfiguration().orientation == Configuration.ORIENTATION_PORTRAIT) {
                    activity.setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
                } else
                    activity.setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
            } else {
                activity.setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_UNSPECIFIED);
                editor.putBoolean( s: "checkbox",  b: false);
                editor.commit(); // commit changes
            }
        default:
            break;
    }
}
```

Figure 28 - Code snippet of the logic behind the screen orientation lock

```
private void changeMapSize(byte data) {
    Log.d( tag: "MapSize",  msg: "Rescaled");
    ((MainActivity) getActivity()).passUserInput((byte) 5, new byte[]{data});
}
```

Figure 29 - Code snippet of the logic behind changing map size

The **onClick** method is fired when one of the buttons or the checkbox is tapped, and the corresponding logic is executed. In the case of one of the two buttons being clicked a user input of plus or minus is sent to the broadcaster which creates the packet with that

information and sends it via BLE. Finally, the checkbox being selected will save the *current state* of the checkbox in a **SharedPreferences** and also lock or unlock the orientation. The **SharedPreferences** are needed in order to store the state of the checkbox through the app's lifecycle.

```
@Override
public void onResume() {
    super.onResume();
    if (pref.getBoolean( s: "checkbox",  b: false) == true) { //false is default value
        checkBox.setChecked(true); //it was checked
    } else {
        checkBox.setChecked(false); //it was NOT checked
    }
    status = ((MainActivity)getActivity()).findViewById(R.id.textView5);
    status.setText(((MainActivity)getActivity()).getStatus());
    Log.d( tag: "UIStatusResume",((MainActivity)getActivity()).getStatus());
    if(status.getText().toString().contains("Not")){
        status.setTextColor(Color.parseColor( colorString: "#FF0000"));
    } else {
        status.setTextColor(Color.parseColor( colorString: "#00FF00"));
    }
}
```

Figure 30 - Code snippet of the logic behind setting the checkbox's status when returning to the view

In onResume (Figure 30) which gets called when the nav bar settings button is selected, the data from the *checkbox* **SharedPreference** is used in order to set the checkbox in the UI. The status text coming from the **MainActivity's getStatus** is set in the fragment's UI and the color is changed depending on whether the AA is broadcasting or not.

6.2 The HoloLens (Unity application)

In this section interesting code snippets and explanations about the implementation can be found, regarding classes on the **HMD** side. This includes the **BLE Receiver** class, the **Decoder** class and the **Mock-up**. As mentioned before, all HMD related code was written in C#, and both the mock-up and Systematic's HMD application were created with the **Unity engine** (thus also using the Unity libraries for C#).

Unity is a cross-platform game engine that allows for **UWP** applications to be developed, the kind of applications that run on Microsoft's **HMD**. The main components of the **Unity Engine** are:

- **The Scene** - the virtual world displayed to the user at a given time
- **GameObjects** - virtual objects that exist in scenes, which the user can visualize
- **MonoBehaviour scripts** - scripts that can be attached to GameObjects allowing them to interact with the user or with each other

6.2.1 BLE Receiver

In **Awake** this class sets up the BLE receiver by instantiating a new instance of the **BluetoothBLEAdvertisementWatcher** class. It also creates the **manufacturerData** of the watcher and sets the **CompanyId** property to the **BEACON_ID** of the BLE sender. Finally, the **Watcher_Received** event is assigned to the watcher's listener and the watcher is started.

Watcher_Received is an async event method which receives the data coming from the BLE sender, then checks if the data is received from the proper source (comparing the BEACON_ID with the manufacturerData of the received advertisement) and at the end passes that data to the Decoder's **Decode**.

```
using UnityEngine;
#if ENABLE_WINMD_SUPPORT
using Windows.Devices.Bluetooth.Advertisement;
using System.Runtime.InteropServices.WindowsRuntime;
#endif
public class Interface_Receiver : MonoBehaviour
{
#if ENABLE_WINMD_SUPPORT
BluetoothLEAdvertisementWatcher watcher;
public static ushort BEACON_ID = 1775;
#endif
    private Decoder eventProcessor;
    void Awake()
    {
        eventProcessor = GameObject.Find("GameManager").GetComponent<Decoder>();
#if ENABLE_WINMD_SUPPORT
watcher = new BluetoothLEAdvertisementWatcher();
var manufacturerData = new BluetoothLEManufacturerData
{
CompanyId = BEACON_ID
};
watcher.AdvertisementFilter.Advertisement.ManufacturerData.Add(manufacturerData);
watcher.Received += Watcher_Received;
watcher.Start();

#endif
    }
#if ENABLE_WINMD_SUPPORT
private async void Watcher_Received(BluetoothLEAdvertisementWatcher sender, BluetoothLEAdvertisementReceivedEventArgs args)
{
ushort identifier = args.Advertisement.ManufacturerData[0].CompanyId;
    if(identifier == BEACON_ID){
        byte[] data = args.Advertisement.ManufacturerData[0].Data.ToArray();
        Debug.Log(data[0]);
        eventProcessor.Decode(data);
    }
}
#endif
}
```

Figure 31 - Code snippet of the logic behind the Bluetooth receiver on the UWP application

In Figure 31, the #if statements are used to ignore the blocks of code that require access to Microsoft's UWP API at compilation time, since the project isn't supposed to run on anything but the Microsoft HoloLens.

6.2.2 Decoder class

The **Decoder** class is responsible for interpreting the data received from the AA into actions to be performed on the HoloLens. It receives data packets from the **BLE Receiver** ("*data*", a byte array variable), which is handled by the **Decode** method.

The **Decoder** class is also where the **SitaWare** application is integrated into the system

```
public byte receivedData = 0;
public byte movementData = 0;
public byte zoomData = 0;
private byte rotateData = 0;
private byte scaleData = 0;
private byte tapData = 0;
private byte[] textData = new byte[36];
```

Figure 32 - Code snippet of the different kind of data the receiver has to process

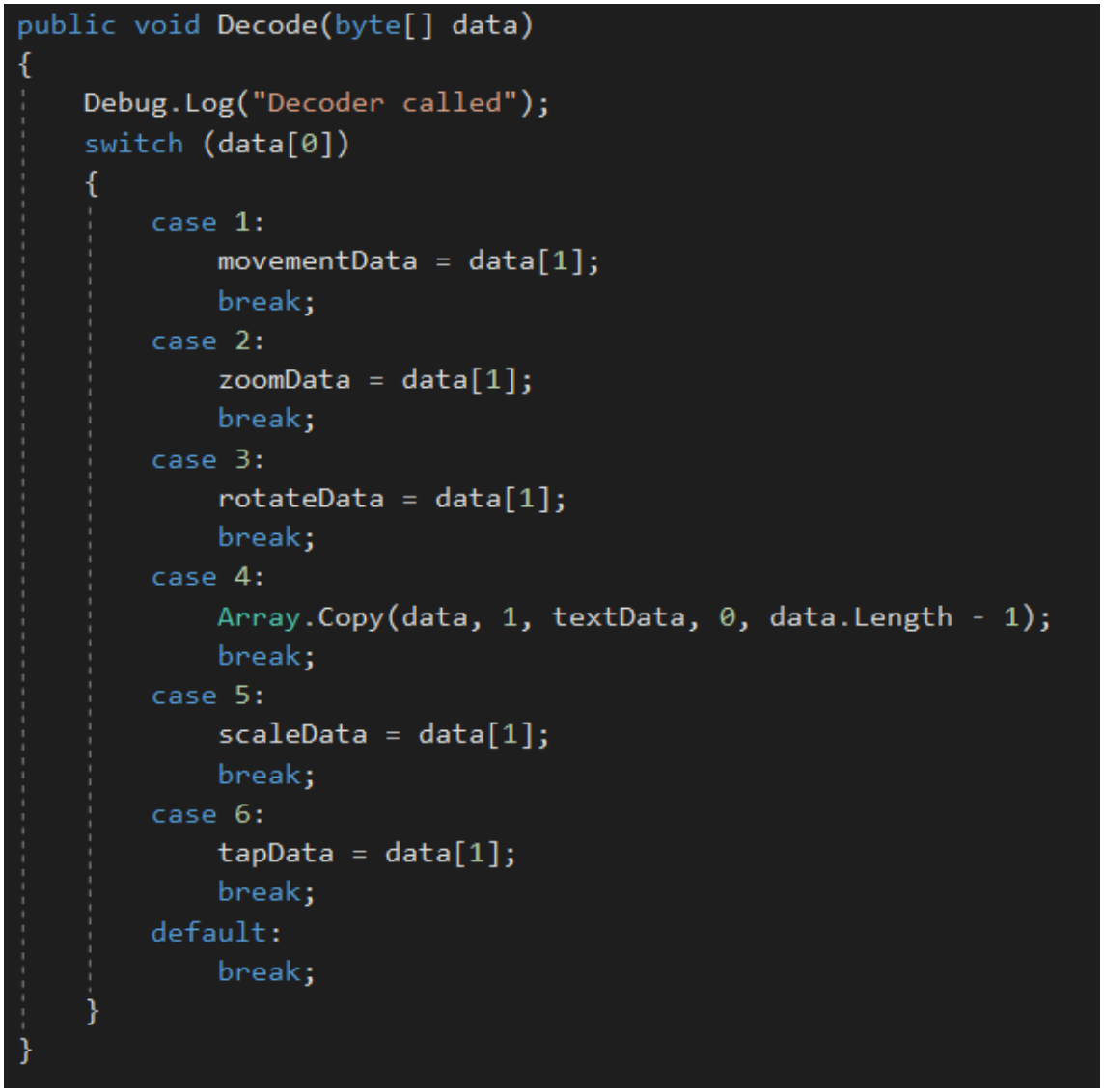
in the means of method calls and references to objects on the **SitaWare** side.

Each functionality has a byte variable assigned to it (e.g. movementData, zoomData, scaleData, etc.). These variables are used for specifying which sub-action to take within a specific functionality (e.g. if zoomData is 1, it means a zoom-in action, if it is 2, it means a zoom-out action). Their default value is 0. *(as seen in Figure 32)*

```
void Update()
{

    if (TiledMap.IsReady && map == null)
    {
        map = GameObject.Find("Map");
    }
    try
    {
        if (terrains != null && terrains.Length == 9)
        {
            scaleData = ScaleController();
        }
        else if (TiledMap.IsReady)
        {
            terrains = GameObject.FindGameObjectsWithTag("TerrainTile");
            MapLocationTracker.Instance.trackedLocationText.text = terrains.Length + "";
        }
    } catch (Exception e)
    {
        MapLocationTracker.Instance.trackedLocationText.text = e.Message;
    }

    if (map != null)
        zoomData = ZoomController();

    movementData = PanMapController();
    rotateData = RotationController();
    tapData = TapSelectionController();
    textData = TextController();
}
```

Figure 33 - Code snippet of the Update() method that gets executed every frame

Each functionality has its own method for manipulating and changing the map (e.g. ZoomController, RotationController, etc.). These methods are running in the **Update** method (Unity's built in method, which is called every time a new frame is loaded) and executed once the value of their corresponding data variable changes from 0. *(as seen in Figure 33)*



```
public void Decode(byte[] data)
{
    Debug.Log("Decoder called");
    switch (data[0])
    {
        case 1:
            movementData = data[1];
            break;
        case 2:
            zoomData = data[1];
            break;
        case 3:
            rotateData = data[1];
            break;
        case 4:
            Array.Copy(data, 1, textData, 0, data.Length - 1);
            break;
        case 5:
            scaleData = data[1];
            break;
        case 6:
            tapData = data[1];
            break;
        default:
            break;
    }
}
```

Figure 34 - Code snippet of the method responsible of filtering the instruction type

The **Decode** method consists of a switch statement, with different cases for the different functionalities. Each case corresponds to a certain functionality. As mentioned in the description in the **Broadcaster**, the first element of *data* (i.e. data[0]) is the ID of a functionality. The switch statement is matching the value of the ID with the different cases (e.g. if ID = 2 a zooming action should be performed). *(as seen in Figure 34)*

Once there is a match with one of the cases, the corresponding data variable of that functionality (e.g. *zoomData*) takes the value of the remaining byte(s) in the *data* object.

This value is relevant in the Update method, since the methods that are used for manipulating the map are "triggered" based on value changes in the different data variables.

```
private byte ZoomController()
{
    if (zoomData != 0)
    {
        RangeSelectionController zoomSelector = GameObject.Find("ZoomSelector").GetComponent<RangeSelectionController>();
        zoomSelector.ZoomChanged(zoomData);
    }
    return 0;
}
```

Figure 35 - Code snippet of the method that handles the zooming functionality

One example of the actual integration of the Sitaware application can be seen in Figure 35, in the **ZoomController** method. Firstly, an if-statement checks whether the corresponding data variable of the zooming functionality (*zoomData*) has been changed from the default value of 0. If so (meaning a pinching gesture has occured on the AA), the necessary code for zooming into the map is going to be executed.

That is done by making a reference to a GameObject (an object in a Unity scene) from the Sitaware application (*ZoomSelector*, the object responsible for handling changes in the zooming level) and calling the necessary method for changing the zoom (*ZoomChanged*). We pass the *zoomData* variable to that method and based on its value a zoom-in or zoom-out action will occur.

6.2.3 TapSelectionController

```
private byte TapSelectionController()
{
    if (tapData == 1)
    {
        Vector3 fwd = uiCamera.transform.TransformDirection(Vector3.forward);
        RaycastHit hit;
        if(Physics.Raycast(uiCamera.transform.position, fwd, out hit))
        {
            if (hit.collider.gameObject.name.Equals("LocationSelector") || hit.collider.gameObject.name.Equals("MapServiceSelector"))
            {
                if (currentDropdown != null)
                    currentDropdown.GetComponent<Dropdown>().Hide();
                currentDropdown = hit.collider.gameObject;
                if (GameObject.Find("Dropdown List") == null)
                {
                    hit.collider.gameObject.GetComponent<Dropdown>().Show();
                }
            }
        }
    }
    return 0;
}
```

Figure 36 - Code snippet of the logic behind the tap selection functionality

TapSelectionController (Figure 36) is responsible for handling tapping in the SitaWare application. In this case it only handles two dropdown menus in SitaWare's solution.

Firstly, an if-statement checks whether the corresponding data variable of the tapping functionality(*tapData*) has been changed from the default value and specifically if it's 1 which means a tapping gesture has occurred on the AA. If so, a forward RayCast is projected from the position of the UICamera and in this case if one of the two dropdown menus' colliders in the UI Canvas is hit, it gets opened. In case a dropdown menu is already opened, tapping would close the said menu if the user tries to open the other menu. Due to time constraints and the complexity of the SitaWare's HMD implementation, the team was able to implement this feature only on some of the current UI elements, as a proof of concept. *(The remaining parts where the feature needs to be implemented are discussed in the Project Future chapter)*

6.2.4 Mock-up

During the early phase of development, a simple mock-up application was made in order to emulate the holographic map. The mock-up consisted of an empty scene

containing a cylinder type *GameObject* that would change its transformation properties (scale, location, rotation) based on the input received from the **AA**.

This approach was taken because of time constraints and in order to achieve a minimum viable product as soon as possible. It also helped the team a lot in understanding how the transformation data gets processed by the Unity engine.

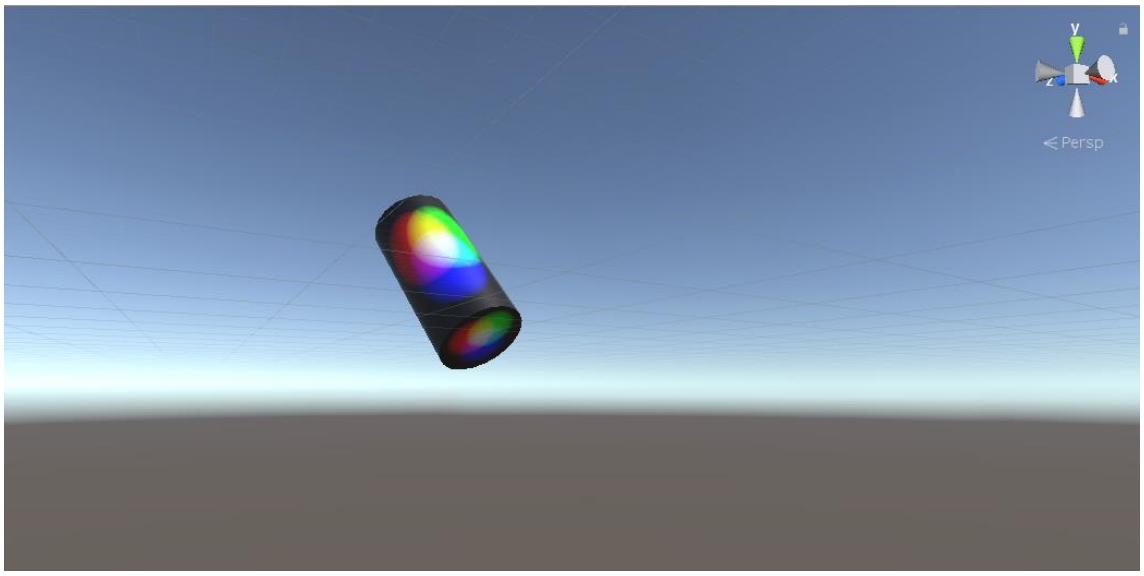


Figure 37 - Screenshot of the mock-up unity scene

The texture applied to the cylinder, in Figure 37, was used in order to have a visual representation of when the team would perform a rotation gesture.

7 Test

This section documents the approach and results of testing.

7.1 Testing the AA

Testing proved to be quite challenging, since the AA is heavily UI based, and the Bluetooth connection is unidirectional. As a testing strategy, white box testing was used to test the system all the way from the UI (user's gestures) to the packets sent by the BLE using the Android Espresso framework.

Android Espresso is a UI based testing framework with an integrated API that can test state expectations, interactions, and assertions. The most useful feature for this project was the included predefined gestures (e.g. tapping, swiping), since 5 of the functional requirements are based on the different kind of gestures the user can perform in order to interact with the HMD hologram. However, the main problem with the framework was the fact that it didn't had multi-touch gesture support. The predefined gestures were limited to scrolling and swiping.

To be able to test the functionalities, that would require the user to use 2 fingers, 2 methods were created for replicating the pinching in/out and rotation gesture. This was achieved by taking advantage of injecting a motion event to the UI controller consisting of two predetermined touch points that have predetermined movement trajectories.

7.2 Testing the HMD Interface

In order to test out the packets the AA is sending via BLE, the tests would decode the last packet created and compare the first byte in the array with the expected instruction ID from the established BLE protocol.

For the Unity packages, including the mock-up application used in the development, the BLE source was also mocked-up and the tests were checking if the GameObject's size/scale/rotation was changing based on the instructions received.

As testing framework, the Unity Test Runner was used, which contains two modes:

- EditMode
- PlayMode

EditMode is used to test logic inside methods, that do not require to be attached to a GameObject in a scene, while PlayMode is used to test interaction between scripts and GameObjects in an empty scene.

Due to the fact, that the transformation data received from the mock-up BLE source needed to be applied to the GameObject in order to check for changes, all the tests were written in PlayMode. *(as seen in Figure 38)*

```
[UnityTest]
public IEnumerator TestRotation()
{
    InitializeScene();
    cyl.transform.localRotation = new Quaternion(30f, 60f, 30f, 0f);
    decoder.Decode(new[] { (byte)3, (byte)1 });
    yield return new WaitForSecondsRealtime(2);
    Assert.AreEqual(0.8d, System.Math.Round(cyl.transform.rotation.y, 1));
    yield return null;
}
```

Figure 38 - Code snippet of one of the automated tests from Unity Runner

In the case of the mock-up, an empty scene was initialized, in which a cylinder type GameObject with predefined properties (scale being set to 0.1 on all 3 axis) and an empty GameObject type object named GameManager was added. GameManager handles the decoding of the mock-up BLE packets. *(as seen in Figure 39)*

```
private void InitializeScene()
{
    if (!cyl)
    {
        var gm = new GameObject("GameManager");
        decoder = gm.AddComponent<Decoder>();
        cyl = new GameObject("Cylinder");
        cyl.transform.localScale = new Vector3(0.1f, 0.1f, 0.1f);
        var mock = new GameObject("MockUp");
        mock.tag = "MockUp";
        mock.AddComponent<MockUp>();
    }
}
```

Figure 39 - Code snippet of how the testing scene gets initialized

7.3 Test coverage

The following functional requirements were covered by the automated tests:

- Panning
- Zoom controls
- Rotating
- Tap selection
- Scaling
- Keyboard input

```
@Test
public void TestBLEStatusText() {
    onView(withId(R.id.gesture_view)).check(matches(isDisplayed()));
    waitFor( seconds: 1);
    onView(withId(R.id.gesture_view)).perform(rotate());
    waitFor( seconds: 1);
    onView(withId(R.id.navigation_settings)).perform(click());
    onView(withId(R.id.textView30)).check(matches(isDisplayed()));
    onView(withId(R.id.textView5)).check(matches(withText("Broadcasting")));
}
```

Figure 40 - Code snippet of one of the automated tests from Android Espresso

Automated tests have the following cycle:

1. The framework waits for the required view to be displayed
2. After the view is displayed it performs an action
3. It waits for 1 second in order to give the data in the background time to update
4. It checks if the updated data matches the expected output
   (as seen in Figure 40)

The automated tests came in handy during the last phase of the project, when the final changes to the AA were made. The newly introduced bugs were detected and fixed immediately.

The last requirement, which wasn't covered by the automated tests (*Communication between Systematic's HMD app and the AA*) was tested by the black box method, passing input using the AA and waiting for the expected changes in the HMD application, since it was the fastest and most reliable way.

7.4 Test Specifications

7.4.1 Features to be tested

1. Test whether the HMD receives instructions properly from the AA
2. Test whether the user is able to pan the map using the arrow keys on the navigation screen
3. Test whether the user is able to zoom in/out by using the pinching gesture on the DTA
4. Test whether the user is able to rotate the map using the two-finger rotation gesture on the DTA
5. Test whether the user is able to use the selection functionality by tapping the DTA in the navigational screen of the AA
6. Test whether the user is able to scale up or down the map from the settings menu
7. Test whether the user is able to input text into the HMD application using the virtual keyboard present in the input screen of the AA
8. Test whether the connection between the system's communication interfaces persists no matter the application's state
9. Test whether the AA prevents the phone from entering standby mode
10. Test whether the AA saves its latest state before losing focus
11. Test whether the AA is responsive towards the orientation of the device.

7.4.2 Features not to be tested

- Test whether the AA is compatible with different HMD implementations

7.4.3 Approach

Manual testing

Functional tests

There will be a set of expected outcomes for these tests, which will be used by the programmer to decide whether the test is a fail or pass.

In the case of the AA not behaving as expected, the case would be considered as fail.

Otherwise the tests are considered as a pass.

Table 2 - Test specifications for features: 1, 2, 3, 4

Number	Pre-conditions
#1	Both the AA and the HMD have Bluetooth turned on
#2	The user is in the navigation view on the AA
Test steps	Test step description
#1	User clicks on the top arrow button. Expected output is that the holographic map pans to the north.
#2	User performs a pinch-in gesture on the DTA. Expected output is that the holographic map zooms in by one iteration.
#3	User performs a clockwise gesture on the DTA. Expected output is that the holographic map rotates 30 degrees clockwise.

Table 3 - Test output for features 1, 2, 3, 4

Test step	Test output Passed/Failed
#1	Passed
#2	Passed
#3	Passed

Table 4 - Test specifications for feature: 5

Number	Pre-conditions
#1	Both the AA and the HMD have Bluetooth turned on
#2	The user is in the navigation view on the AA
Test steps	Test step description
#1	User aims the cursor over a dropdown list and single taps the DTA. Expected output is that the dropdown list expands.

Table 5 - Test output for feature 5

Test output

Test step	Test output Passed/Failed
#1	Passed

Table 6 - Test specifications for feature: 6

Number	Pre-conditions
#1	The AA is running
#2	The UWP is running
#3	The user is in the setting view
Test steps	Test step description
#1	User clicks on the scale down button. Expected output is that the map scales down by one iteration.

Table 7 - Test output for feature 6

Test step	Test output Passed/Failed
#1	Passed

Table 8 - Test specifications for feature 7

Number	Pre-conditions
#1	The AA is running
#2	The UWP is running
#3	The user is in the input view
Test steps	Test step description
#1	User inputs a valid exaggeration level (an integer between 0 and 12).
#2	User clicks the send button. Expected output is that the exaggeration level of the map's terrain changed.

Table 9 - Test output for feature 7

Test step	Test output Passed/Failed
#1	Passed
#2	Passed

Non-functional tests

Table 10 Test specifications for features: 8, 9

Number	Pre-conditions
#1	The AA is running
#2	The UWP is running
Test steps	Test step description
#1	User awaits up to 1 minute, not interacting with the map. Expected output is that the phone's screen does not lose focus of the AA.

#2	User goes to navigation screen. Expected output is that the AA is displaying the navigation screen.
#3	User taps the top arrow button. Expected output is that the map pans north.
#4	User exists the AA and launches it again, then user taps the top arrow button. Expected output is that the map pans north.

Table 11 - Test output for features 8, 9

Test step	Test output Passed/Failed
#1	Passed
#2	Passed
#3	Passed
#4	Passed

Table 12 - Test specifications for features: 9, 10

Number	Pre-conditions
#1	The AA is running
#2	The UWP is running
#3	The user is navigation view
Test steps	Test step description
#1	User tilts the phone 90 degrees into landscape mode. Expected output is that the UI of the phone changes to match the current phone orientation.
#2	User restarts the AA. Expected value is that the AA keeps the screen orientation from when it stopped working.

Table 13 - Test output for features: 10, 11

Test step	Test output Passed/Failed
#1	Passed
#2	Passed

7.4.4 Item pass/fail criteria

Automated tests with Android Espresso and Unity Test Runner:

There will be a set of expected results for these tests, which will be used by the tool to decide whether the test is fail or pass.

All the test cases flagged as unexpected fail are considered as fail.

All other tests are considered as having passed.

Table 14 - Automated tests overview (Android)

Test Name	Test output Passed / Failed
Test BLE Status Text in Settings View	Passed
Test Map Panning	Passed
Test Text Input	Passed
Test Map Rotation	Passed
Test Map Scaling	Passed
Test Zooming Out	Passed

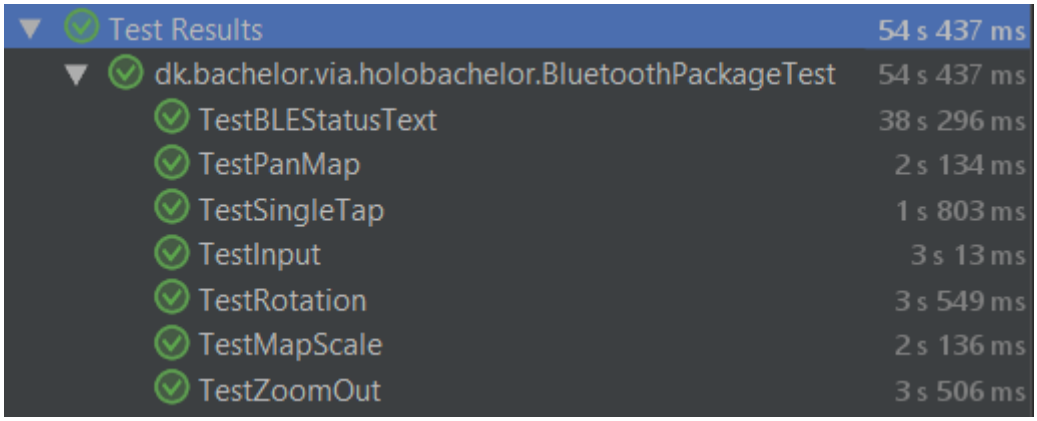


Figure 41 - Android Espresso view of tests status

Table 15 - Automated tests overview (Unity)

Test Name	Test output Passed / Failed
Test the panning instruction on the mock-up GameObject	Passed
Test the rotation instruction on the mock-up GameObject	Passed
Test the zooming instruction on the mock-up GameObject	Passed

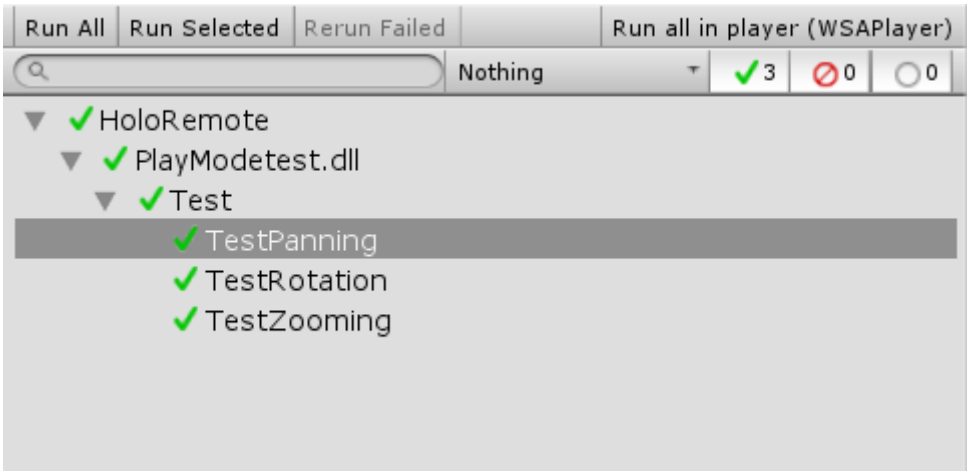


Figure 42 - Unity Test Runner view of tests status

8 Results and Discussion

This section presents the outcome and achieved results of the project.

8.1 Achieved results:

- Suitable and functioning UI for the AA (menu bar, DTA, text input, settings menu for user preferences)
- Sending executable commands in the form of data packets from an Android device to Microsoft HoloLens through Bluetooth Low Energy, based on user input
- Gesture recognition system in the AA
- Sending a string from the AA to the HMD, interpreting and processing it afterwards
- Integration of Systematic's HMD application into the system
- Responsive layout for the AA (portrait and landscape mode)

8.2 Discussion

The outcome of the project was a working prototype of the system the team had in mind: An Android application capable of interacting with a Mixed Reality HMD application, controlling it remotely.

9 Conclusions

This section compiles the results from each section in the report.

9.1 Requirements

After thorough testing the team has concluded that most of the functional requirements of the project have been fulfilled:

- The communication between the AA and the HMD is established automatically
- The navigational functionalities (panning, zooming, selecting and rotating) are operational
- The system is also capable of sending and interpreting string data (text input).
- The functionalities of the Settings menu (scaling, orientation lock) are also working as expected.

Furthermore, most of the non-functional requirements are also fulfilled:

- *Availability*: the connection between the AA and the HMD is persistent whether the AA has focus or runs in the background
- *Performance*: the AA prevents the device from entering standby mode, moreover, the AA saves its last state before losing focus (e.g.: which fragment was opened)
- *Usability*: the AA supports both portrait and landscape mode
- *Interoperability*: due to time and resource constraints the team didn't have a chance to fulfill the requirement of making the AA compatible with different HMDs

9.2 Analysis

As previously mentioned, the SitaWare system requires precise controls in order to be used efficiently. After testing the system with the controls provided by the AA interaction with the holographic objects seemed to be far less cumbersome in comparison to the existing controls of the HoloLens (hand gestures/clicker). Thus, it can be concluded that the Android remote controller makes a difference for the better in terms of user experience when it comes to using mixed reality HMDs.

9.3 Design

Due to the simplicity of the project and the lack of need to store the data in a persistent manner, the use of a software architecture was thought to be unnecessary. Because of that, the code was implemented in a modular way, not following any of the architectures, but still trying to respect the OOP's SOLID principles.

The mixed reality HMD chosen to be used for the project is Microsoft HoloLens, since this is what the current SitaWare MR implementation is based on. The Android operating system was selected, since it is a reliable technology that has an easily customizable user interface.

As for the three technologies discussed for the connection, Bluetooth Low Energy was the most suitable one for the project's needs since it has low latency, it provides a local connection between the devices, saves battery lifetime and it allows data transfer to a sufficient extent.

Finally, in order for the HMD interface to understand the instructions the AA is sending, a custom protocol was established.

9.4 Implementation

As discussed before, the implementation of the system consisted of two major parts: creating the broadcaster (AA) and receiver (HMD) side

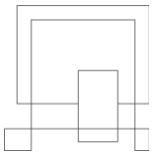
Creation of the **AA** proved to be the bigger task, since the team had to start from scratch. However, thanks to the team's knowledge about Android Studio and Java the development process went smoothly. Setting up the Broadcaster class and implementing the necessary logic for sending different data packets for different user gestures proved to be easier than expected.

Naturally, there were also several challenges on the way, but all of them were managed with relative ease. One of the bigger challenges during the development of the AA was ensuring support for both landscape and portrait mode in the application. Changing between the two modes created numerous bugs in the application, though eventually all of them were fixed. Using fragments was also a bigger task to undertake for the team, as the none of the members had any experience with using them. In the end the integration of all three fragments into the main activity turned out to be a highly operational solution.

The **HMD** side of the implementation was less time consuming, since the HMD application itself was already provided by Systematic. Therefore, the only task was to create a receiver and a decoder class, both turning out to be completely operational once we have integrated the SitaWare software into our system. It was also helpful that each of the team members were familiar with the workings of the Unity engine and C#.

9.5 Test

The testing approach conducted for most of the requirements was white-box testing due to the fact that the AA was heavily UI based. Testing user interactions was simple except for the gestures that would require the user to use two fingers. For these cases, some methods were created that would emulate the user's two finger inputs.



As for the HMD, the mock-up object's transformation was tested based on the data received by the interface, using the white-box testing approach.

The last requirement, the connection between the AA and the HMD, was tested using the black-box method.

All tests passed in the end, proving that all the requirements were implemented in the correct manner.

10 Project future

From a technical viewpoint the project consists of an AA which has one activity and three fragments, which are being recycled depending on the user's interactions. Furthermore, an HMD app is present which establishes a connection to the AA and decodes the user's interactions on the AA, and thus converting them to HMD interactions.

The current implementation of the system provides a unidirectional connection between the devices, therefore, an important aspect to keep in mind during the next iterations could be to find a way for the HMD to communicate back to the AA. That could be done by making the BLE connection bidirectional, despite the fact, that at the moment making a persistent BLE connection between the two devices might not be achievable due to the differences between the platforms.

Another improvement could involve looking into the latency between the devices, and also making sure, that there are no dropped (the HMD should pick up all the packets transmitted by the AA). While developing the application the team has noticed, that the window for scanning might not be the most optimal one, some packets not being delivered, or other packets getting delivered twice. A possible fix for this might be increasing the time of the scan interval and at the same time adding a unique identifier for each packet. This way, if the HMD receives two consecutive packets, it would execute only the first one.

Moreover, the AA could also be modified in a way to update the HMD as soon as it detects a gesture, and not when the user stops interacting with the DTA. This could be done by finding a way to stop the BLE thread from crashing when it broadcasts too many packets in a short time interval.

The selection tap gesture could also be implemented to interact with the remainder of the UI elements present on the canvas on top of the holographic map and also to replace the current map positioning feature by having the same placement checks as the current implementation.

The way the HMD interface was implemented could help in repurposing the remote controller for other applications with minimal code changes.

11 Sources of information

Anon 2018. *4. GATT (Services and Characteristics) - Getting Started with Bluetooth Low Energy [Book]*. [online] Available at: <https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html> [Accessed 6 Oct. 2018].

Anon 2018. *Bluetooth Low Energy Scanning and Advertising*. [online] Available at: <http://dev.ti.com/tirex/content/simplelink_academy_cc2640r2sdk_1_12_01_16/modules/ble_scan_adv_basic/ble_scan_adv_basic.html?fbclid=IwAR2kzM8D1ofPYDjoUWArUK2shCcn7f0y2FQTmNO5yc2vK8TjTk5cJWCU-wQ> [Accessed 6 Oct. 2018].

Anon 2018. *Fragments | Android Developers*. [online] Available at: <https://developer.android.com/guide/components/fragments> [Accessed 15 Nov. 2018].

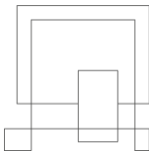
Anon 2018. *Gestures - Mixed Reality | Microsoft Docs*. [online] Available at: <https://docs.microsoft.com/en-us/windows/mixed-reality/gestures> [Accessed 30 Sep. 2018].

Anon 2018. *Introduction to Activities | Android Developers*. [online] Available at: <https://developer.android.com/guide/components/activities/intro-activities> [Accessed 15 Nov. 2018].

Anon 2018. *Microsoft HoloLens | The leader in mixed reality technology*. [online] Available at: <https://www.microsoft.com/en-us/hololens> [Accessed 20 Sep. 2018].

Anon 2018. *MoSCoW method*. [online] Available at: <https://en.wikipedia.org/wiki/MoSCoW_method> [Accessed 11 Dec. 2018].

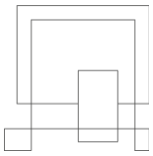
Anon 2018. *MVC, MVP, MVVM Design Patterns with Godfrey Nolan - YouTube*.



[online] Available at: <https://www.youtube.com/watch?v=JV63czrUpbI&feature=youtu.be%5C> [Accessed 25 Oct. 2018].

Anon 2018. *Systematic | Defence*. [online] Available at: <https://systematic.com/defence/> [Accessed 10 Oct. 2018].

Cho, K., Park, W., Hong, M., Park, G., Cho, W., Seo, J., Han, K., Cho, K., Park, W., Hong, M., Park, G., Cho, W., Seo, J. and Han, K., 2014. Analysis of Latency Performance of Bluetooth Low Energy (BLE) Networks. *Sensors*, [online] 15(1), pp.59–78. Available at: <http://www.mdpi.com/1424-8220/15/1/59> [Accessed 05 Dec. 2018].



12 Appendices

Appendix A – Project Description

Appendix B – Use case & activity diagrams

Appendix C – Class & sequence diagrams