FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Virtual Reality Menü Interaktion mit einer Smartwatch

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medieninformatik

eingereicht von

Florian Schuster, BSc
Matrikelnummer 1025700

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Priv. Doz. Mag.rer.nat. Dr.techn. Hannes Kaufmann
Mitwirkung: Kolleg. Iana Podkosova, BSc MSc

Wien, 11. August 2016

Florian Schuster	Hannes Kaufmann

Virtual Reality Menu Interaction with a Smartwatch

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Media Informatics

by

Florian Schuster, BSc
Registration Number 1025700

to the Faculty of Informatics
at the TU Wien

Advisor: Priv. Doz. Mag.rer.nat. Dr.techn. Hannes Kaufmann
Assistance: Kolleg. Iana Podkosova, BSc MSc

Vienna, 11th August, 2016

Florian Schuster	Hannes Kaufmann

Erklärung zur Verfassung der Arbeit

Florian Schuster, BSc
Heigerleinstrasse 36/2, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. August 2016

Florian Schuster

Acknowledgements

I would like to express my gratitude towards everyone who supported me during my studies and the creation of my master thesis. Special thanks to my advisors Prof. Kaufmann and Ms. Podkosova, MSc. for their superb assistance and support. Thank you for your guidance and for giving me the possibility to write a thesis on this topic.

And of course I would like to thank my parents who did not only raise me, but also gave me the chance to study in a field of my choice. Likewise I want to thank my girlfriend who always stood by my side.

Thank you.

Kurzfassung

In immersiven virtuellen Umgebungen werden Benützer in Virtual Reality (VR) Simulationen versetzt. In solchen Umgebungen werden Menüs gebraucht, um den Systemzustand zu ändern, verschiedene Tools (Werkzeuge) auszuwählen oder bestimmte Kommandos zu geben. Allerdings kann es bei der Interaktion mit solchen Menüs zu Schwierigkeiten kommen, da die Benützer Head Mounted Displays (HMDs) tragen und somit nicht in der Lage sind, physische Eingabegeräte zu sehen. Deswegen muss die Interaktion mit denselben intuitiv und einfach zu erlernen sein.

In dieser Diplomarbeit wird eine Möglichkeit vorgestellt, ein VR Menü mit der Hilfe einer Smartwatch über ein Wireless Local Area Network (WLAN) zu steuern. Dafür wird ein Plugin für die Unreal Engine 4 (UE4) Spiele-Engine implementiert. Dieses Plugin enthält Komponenten, die mit einer Smartwatch gesteuert werden können. Weiters wird eine Android Wear Applikation erstellt, die es ermöglicht, mit Hilfe von Touch-Interaktion die Komponenten zu steuern.

Die Applikation und das Plugin werden letztendlich mit der Hilfe von einem der vier erstellten Test-Menüs einem Leistungstest unterzogen, um etwaige Auswirkungen auf die Hardware zu entdecken. Die Ergebnisse zeigen, dass die Leistung durch die Implementationen nicht eingeschränkt wird.

Abstract

In immersive virtual environments users are placed inside virtual reality simulations. In such systems, menus are needed to change the system state, change tools or issue commands. However interacting with these menus in VR can be cumbersome, since input devices are not seen by the user due to the HMD. Therefore interaction needs to be simple and easy to adopt to.

This diploma thesis presents a possibility to navigate VR menus with the help of a smartwatch over a WLAN. For this, a plugin for the UE4 game engine is implemented which contains components to create a smartwatch controllable menu. Furthermore an Android Wear application is designed and implemented to enable plug-and-play navigation of the menu via touch interaction.

The resulting applications and the plugin are then tested to detect any issues regarding performance impact, with the help of one of four created example menus. The results reveal that performance is not affected considerably.

Contents

CHAPTER 1

Introduction

VR refers to an artificially created environment that simulates sensory experience, such as sight, touch or hearing to its users. With HMDs such as the Oculus Rift (CV) and the HTC Vive being released on the market, Virtual Environments (VEs) can now be explored by a broad audience of consumers at home. These two HMDs work with optical tracking which allows the user to walk and interact inside a small VE of approximately 4x4 meters.

Apart from VEs that work with rather small tracking areas [34, 69] there are also those that extend large areas and offer tracking of several users [68, 25, 40, 56]. In such systems the users need some sort of system control which makes it possible to issue commands, change modes or choose tools. This can be achieved by providing users with a virtual in-game menu with which they can interact.

However there are certain problems associated with menus in VR. The first one is the required input method. Conventional input methods for computer systems such as keyboards, mouses, joysticks or controllers are impractical since they cannot be seen by the users due to the HMDs. Such interaction devices are also cumbersome because they are heavy and have too many different buttons which might confuse users, since their eyesight is temporarily lost due to the HMD. A potential input device would have to be lightweight and simple to use, to enable efficient interaction with the menu.

Another problem with VR menus is that there is no general approach to their development. There are no official guidelines which tell developers where a menu has to appear, how the user has to interact with them or if they should be a menu in the traditional sense at all. General Human Computer Interaction (HCI) guidelines [52] recommend rules like consistency of the system, appropriate feedback or constructive error messages that should be considered for VR menus. However, they focus on using mouse and keyboard as interaction devices and do not consider the complexity that is added when interacting in a three dimensional VE.

1.1 Motivation

The motivation of this thesis is to enable user interaction with VR menus via the touch input of a smartwatch. It should be simple for the user to interact with as well as simple for the developer to create a VR menu that can be navigated and controlled with a smartwatch. Ultimately, the construction of the VR menu should work directly inside the graphical editor of a game engine.

There have been no publications regarding this topic yet which makes it very intriguing and interesting.

1.2 Problem Statement

In VEs such as explained in [68, 25, 40, 56], the users move around with a HMD and have a world to explore which must not necessarily resemble the real world. Thus they might be disoriented or too distracted to control additional menus with complex interaction techniques. A smartwatch could be an interaction tool which provides a solution for these problems. It is simple to use with touch gestures, fixated on the wrist, always in reach of the users hand, and has a nearly unnoticeable weight.

However there is no research as well as there are no implementations of the combination of smartwatches and VR menus. Even in well established game engines, such as Unity [18], UE4 [19] and CryEngine [9], support for smartphone-based or smartwatch-based menu or game interaction is simply not offered. This means that the whole system of communication between the smartwatch and the game application has to be designed. Additionally the user's input has to be correctly registered and then mapped to specific events in the game and the virtual menu.

1.3 Expected Result

The goal of this thesis is the design and implementation of a correlating system of two components. First, a plugin for the UE4 game engine is developed which contains components that can be used by a developer to create a smartwatch controllable VR menu in the graphical editor. Secondly, an Android Wear [4] application is created which enables users to easily control the constructed VR menu. The communication between the Wear-UE4 system should be over a WLAN and the setup should be as straightforward as possible.

1.4 Methodological Approach

First, the correlation in the Wear-UE4 system is designed. It is determined whether there should be a connectionless or a connection-oriented interplay between the UE4 plugin and the smartwatch application.

Afterwards, the Android smartwatch application is created to allow user input. Also a smartphone application with a background service is required to have access to a

WLAN to transmit data from the phone to the UE4 plugin. The smartphone service and the smatwatch application need to communicate via a bluetooth-tethered connection to transmit touch input data from the watch to the phone.

Then, the UE4 plugin is implemented with different Unreal Motion Graphics (UMG) menu interface components which can be used in the graphical editor of the engine. Afterwards the connection with the Android device is established with the method that has to be determined in the first step.

Finally, the applications is put to the test with the construction and usage of multiple menus.

1.5 Thesis Outline

Chapter 2 discusses the theoretical background and state-of-the-art methods of VEs, VR menus and interaction with smartwatches. In chapter 3 the designing process of the UE4 plugin and Android application, as well as their connection via a WLAN are described. It also elaborates platforms, languages, tools and designing methods used to design the Wear-UE4 systems. Chapter 4 then goes into detail on the implementation process of both applications. It also describes how both applications can be used by users and developers. In chapter 5, results concerning resource usage are elaborated on and some example VR menus are presented. The last chapter, 6, concludes the thesis.

CHAPTER 2

State of the Art

2.1 Immersive Virtual Reality Systems

In this chapter state-of-the-art immersive VR systems, meaning systems in which users can walk and interact are presented. Afterwards, fundamentals of user interaction in such systems are discussed and finally research on smartwatch interaction in general is presented.

2.1.1 Systems

In [68], Wallner et al. present *The HIVE*, a huge immersive VE. In *The HIVE* an eight-camera outside-in infrared Precision Position Tracker system is used to cover the whole area with a precision of 0.63 cm and reduce possible occlusion problems. The position data acquired by the tracking system is sent wirelessly to the rendering computer worn on the back of the user. This backpack weighs about 9.8 kilograms and also contains an Inertial Measurement Unit (IMU) to track the users orientation. The different forms of data are fused and then used to continually update the user's point of view. The HMD used is the NVIS nVisor SX with a stereo display with the resolution of $1,024 \times 1,280$ for each eye. Also a position tracking computer for the camera system, a WLAN hub and a control room for the graphics workstation that monitors and maintains the VE server state are needed in the setup at a fixed location. The system enables untethered tracking in a physical space of 570 m^2, but can only track the head positions of several users or the body parts of one user at the same time.

Another immersive VR system is described in [25] by Bachmann et al. In *Going anywhere anywhere*, the authors propose a system that is completely portable. The setup consists of a couple of ultrasonic Transducers, a HMD, a head mounted inertial/magnetic orientation sensor, a Global Positioning System (GPS) antenna, a wearable rendering unit, a video control unit and a pair of foot mounted inertial/magnetic sensor modules. Portability is achieved by combining Redirected Walking (RDW), Selfcontained Inertial

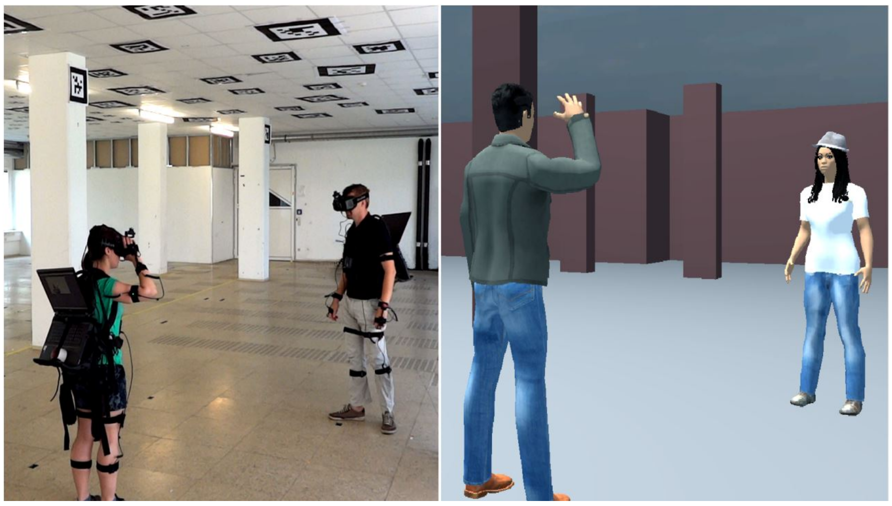


Figure 2.1: Two users interacting in the ImmersiveDeck.

Position Tracking (SCIPT) [72] and the use of an ultrasonic ranging system to Simultaneously Localize and Map (SLAM) within an unfamiliar tracking area. Due to its portability, the system is usable in any flat area such as an athletic field, a parking lot or a backyard, not requiring a permanent infrastructure as a result. Users can navigate through a VE with an unlimited size by walking naturally without becoming aware of the physical limits of the area they are in. This is achieved with RDW which imperceptibly steers the user away from obstacles such as boundaries and walls, or other users by judiciously rotating the virtual scene about the user [59].

In [56], the preliminary results of a low-cost multi-user immersive VR system called *ImmersiveDeck* (Fig. 2.1) are presented by Podkosova et al. The system uses inside-out optical head tracking, in contrast to The HIVE, coupled with a low-cost motion capture suit to fully track the body of several users and allow complex multi-user and object interactions in a 200 m^2 area. Due to the nature of marker tracking, the size of the tracking area is only limited by the amount of ID markers and can be scaled up easily and inexpensively. To achieve the requirement of low-cost, the authors only used off-the-shelf hardware: A processing laptop that is strapped to the back of the user, a motion capture suit and an Oculus Rift Developer Kit 2 (DK2) with an attached fish-eye tracking camera and headset. The computation of input data and rendering of output information for the user's HMD are performed locally which leads to minimal update latency between movement and resulting action in the VE. The data, consisting of information from the fish-eye camera, the sensors of the Oculus Rift and the motion capture suit, is fused and then distributed over a WLAN to the server and to the other clients. The fish-eye camera tracks the eighty square ID markers that are distributed evenly on the ceiling of the whole area. Because it has a field of view of up to 190°, it can track the markers even when the lens is not pointing directly at the ceiling, thus providing the user's position and pose in the VE. The Oculus Rift tracking data is used to set the rotation of the user's point of view. The motion capture suit on the other hand tracks the movement of the users limbs with the help of eleven IMU sensors. In the beginning of each session the suit has to be calibrated briefly by taking a pre-defined pose that corresponds to the avatar's pose. Haptic feedback is provided by real objects that are also tracked by the system: Big objects such as tables or chairs are tracked globally with multi-marker tracking by a camera connected to the server. Small objects such as boxes are tracked locally with an Android smartphone which is also mounted to the HMD of the user.

An interesting commercial idea for VR entertainment is offered by a company called *The Void* [17], or the Vision of Infinite Dimensions. The company will open an enternainment center in late summer 2016 in the suburbs of Salt Lake City. It consists of seven 18 m^2 rooms that allow multiple users to interact in different VR scenarios. The developers use their own technology, among other things their own HMDs, controllers and walls that change texture, temperature and move to simulate different environments and evolve naturally as you interact with them. Users are guided through multiple VEs with the help of RDW and the simulation of other transportation methods such as for instance elevators [12].

When looking at the advancements of immersive VR systems over the last few years, it is safe to say that the overall price of these systems gradually decreases and thus availability for the public increases. The authors of [40] demonstrate an example of the rapid decline of cost in VR: A system from 2010, based on the system from [25], weighed approximately 11 kg and cost about 45000$ to assemble. In 2013 the authors build a system with comparable quality which weighed only 2.3 kg and cost about 1300$.

VEs have been used as an effective instrument of communication [28], education [70], entertainment [26], social interaction [29], training [62] and of course research [68]. Such VE systems and the immersive systems described before, will find a more widespread use in the future. Thus methods to control and interact with them are needed. Especially new methods to change the game state are required, since they are arguably the most abstract to implement.

2.1.2 User Interaction in Virtual Reality Systems

VEs come in many different forms, one common denominator however is the user. The user needs to interact with the system to accomplish a given task via the interface and for that there are numerous methods.

In [32], Bowman explains that there are two types of components used to interact with a 3D interface: hardware components and software components. Hardware components are input and output devices such as for example controllers, HMDs and displays. Software components are control-display mappings such as for instance ray casting or virtual hand control. With both types of components the choice that the developer has to make is whether the control should feel either *natural* or *magical.* Interaction can work exactly like in the real world, such as natural walking through a VE. Interaction can also work *magical* which means giving users new possibilities that they would not have in the real world. An example is the Go-Go Technique [57, 30] which extends the length of the virtual arm of the user by a coefficient after a certain distance.

However the cognitive overhead that is required for non-natural interaction techniques can distract or even annoy the user, so they have to be used carefully. It is not easy to find a 3D interaction technique that is efficient enough, provides a good usability and is also useful for the user for a specific task. That is the reason why there are no constraints, no guidelines and no standards. Furthermore interacting in three dimensional space is more complex due to the spatial component.

The *Universal Interaction Tasks* presented by Bowman assort all 3D interaction methods into the following categories: Selection, Manipulation, Navigation, System control and Symbolic input. The interaction techniques that are most relevant for this thesis and directly connected to VR menus, are *Selection*, *System control* and *Symbolic input*.

Selection tasks are performed to pick *n* objects from a set with goals such as to indicate action on them or make them active. This can be done by either pointing with techniques such a the Go-Go Technique or World-In-Miniature (WIM) [65]. It can also be done by naming the objects through speech recognition.

System control tasks are performed to change either the mode of interaction or the state of the system. Techniques include different kinds of menus which will be discussed in section 2.2.1, voice commands which include speech recognition or spoken dialogue systems and gestural commands.

Symbolic input shows a large overlap with *System control* tasks, as it provides methods to input numerical or symbolic data and enables precise labeling. Classic input methods include keyboards or numpads but there are also pen-based [47, 58], smartphone-based [36], speech-based [50] and gesture-based [45] techniques.

Manipulation tasks modify an object's property such as the position, orientation or shape. Manipulation metaphors include natural interaction techniques such as the simple virtual hand or the HOMER (Hand-centered Object Manipulation Extending Ray-casting) technique [30].

Navigation tasks consist of the motor travel component and the cognitive wayfinding component. The travel component is in charge of setting the position and orientation of the user's viewpoint in the VE. The most basic technique here is natural walking which is used in many large-scale immersive VR systems. The wayfinding component of a *Navigation* task is defined as the cognitive process of finding a path through a VE using spatial knowledge [35].

In general when creating or deploying different interaction approaches, the developer has to think about the artistic approach, such as aesthetics, adaption and intuition of the user, as well as the scientific approaches, such as performance requirements and formal analysis of the system and try to find a balanced approach.

2.2 Virtual Reality Menus

As mentioned before, VR menus are a method to provide the user with *System control*. In fully immersive VR systems, users should not have to take off their HMDs to change the settings or state of the system. System control has to happen inside the application and this is where graphical menus come into play.

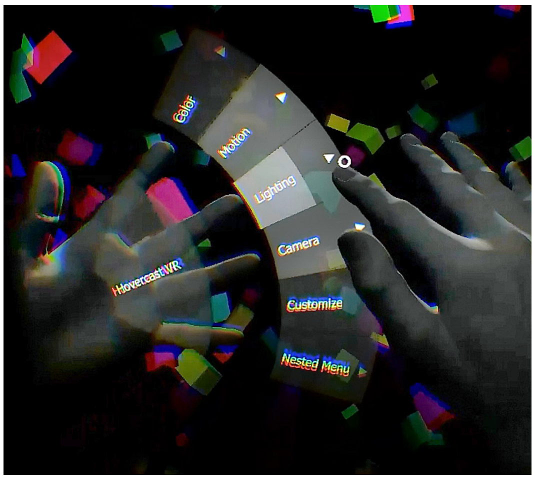
2.2.1 Types of Menus

In an article from WEareAR [21], three major design directions are proposed for menus in VR applications: Skeuomorphic Menus, Flat Menus (Mapped on Geometry) and 3D Menus.

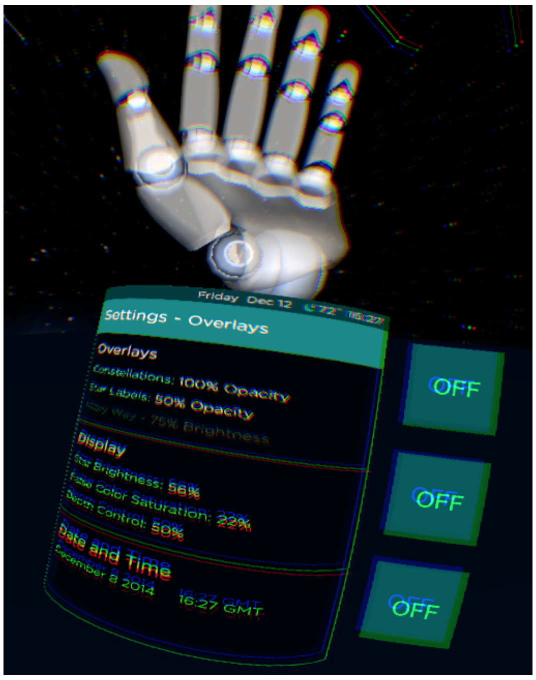
Skeuomorphic Menus use references to real-world objects or experiences to help users understand how the interaction works. This design approach serves as a bridge into new technology since the user has a mental model of how interaction with real world objects work. An example of skeuomorphic design would be a wardrobe in a Role Playing Game (RPG). Users could look through the shelves of the wardrobe for clothes that they want their virtual avatar to wear. In an VE there would be no need for the shelves, theoretically it is possible to let the clothes appear out of thin air in front of the user. However to achieve simplicity and new user adaption, the shelve metaphor is used. Another example would be a computer or a smartphone that acts as a menu, which is for instance employed in games such as Grand Theft Auto 5 or Fallout 4.

Flat Menus (Mapped on Geometry) are menus that are mapped onto environment geometry. They act like interactable textures that are, for example, placed on the surface of a wall, a window or the bodypart of the avatar. An example is the Oculus Home application which acts as a VR store for the Oculus Rift and is mapped onto the inside of an invisible spherical surface to look like a curved screen. Another example is the fader widget used for the iOrb [60] interface. The iOrb is a ball-like device which enables three-dimensional input for applications. The user rotates the iOrb to select the desired element within the fader widget, which is basically a radial menu. Generally these flat menus require previous knowledge of usage or adaption time since users might not know the predefined position in the VEs. An approach is to have the menu appear directly on the user's avatar. An example is the TULIP menu [33] that is drawn at the end of the user's fingers and controlled with Pinch Gloves. Each finger manages one menu point and when the finger is contracted, the appropriate point is selected. The problem that a user only has five fingers and thus making only five options available is regarded in a newer but similar approach to the TULIP menu: the Hovercast VR interface [10] by Leap Motion. It is an arc-shaped menu that extends from the fingertips (Fig. 2.2a) of one of the user's hands. The other hand is used to navigate through the hierarchical menu items which include buttons, check boxes, radio buttons and sliders. The tracking is done with the Leap Motion controller.

3D Menus are interfaces that use the third dimension as additional input for the user. They are especially hard to create because there are no specific design rules, as opposed to two dimensional interfaces. An example for a 3D menu is the ring menu described in [37]. 3D Items are arranged on a portion of a circle and selected by rotating the wrist with the help of the Wand of the Intersense IS-900 System. Another quite interesting approach is the Arm HUD [8] by Leap Motion. This interface is, same as the Hovercast interface, controlled with the Leap Motion controller and attached to the hand of the



(a) Hovercast



(b) Arm HUD

Figure 2.2: VR Interfaces by Leap Motion.

user like a smartwatch (Fig. 2.2b). The Arm HUD changes its functionality based on the orientation of the user's arm. Looking on the menu as if checking a wristwatch, opens up a different menu than looking on the inside of the wrist.

In conclusion, the developer's choice of the menu's design direction should depend on the dimensionality of the available input device. *3D Menus* for instance, cannot be used efficiently with two dimensional controllers such as a touchpad or simple cursor mechanics because they do not cover the third dimension. For that purpose other tracking devices such as the Leap Motion, Wands or Pinch Gloves have to be used. Two dimensional menus however can be controlled with three dimensional input devices, as we have seen with the TULIP and the Hovercast menu. However this additional unused dimension might confuse novice users as is explained in the next section.

2.2.2 Guidelines

User Interface (UI) guidelines are employed by every major company: Android uses their Material Design Guidelines [6], Apple their iOS Human Interface Guidelines [7] and Windows their Universal Windows Platform Guidelines [22]. Also major VR brands have their own guidelines, for example Oculus [14] and Leap Motion [11]. Yet there are general design heuristics for HCI that also apply to VR menus. In [52], Molich and Nielsen provide principles of interaction design:

1. Simple and Natural Dialogue
   Interfaces should not contain any irrelevant information.
2. Speak the User's Language
   Interface concepts should be user-oriented and not system-oriented.
3. Minimize the User's Memory Load
   Users should recognize interface components and not recall them.
4. Be Consistent
   Interface concepts should always mean the same thing to the user, even in subsystems.
5. Provide Feedback
   The user should be informed appropriately of the system's current state.
6. Provide Clearly Marked Exits
   Interfaces should always provide an escape possibility for the user.
7. Provide Shortcuts
   Interfaces should use accelerators that help (experienced) users to navigate quicker.
8. Error Prevention
   Try to eliminate error-prone conditions or inform the user about them.
9. Provide Good Error Messages
   If an error does occur, interfaces should provide users with precise and constructive error messages.

These however are mere heuristics and by no means strict guidelines. For an interface designer, guidelines should be a starting point, proposed, to help ensure that users can interact with the interface more intuitively.

When looking for appropriate UI heuristics for VR menus, the three types have to be discussed separately.

Skeuomorphic Menus provide the user with a mental model of how the menu works and thus possess inherent guidelines. An example is a skeuomorphic computer menu, as mentioned in section 2.2.1. The developer can adapt his menu to an already existing design, for example a Graphical User Interface (GUI) of a PC. Nevertheless, general rules for interfaces such as those proposed by Molich and Nielsen still apply.

Since *Flat Menus (Mapped on Geometry)* are two dimensional interfaces, guidelines do exist. There are numerous publications on flat UI design: [63, 49, 44, 39, 55]. In [13], the Nielsen Norman Group released a list of guidelines for website and application menus:

1. Make It Visible
   - Menus should have an appropriate size, depending on the output device.
   - Menus should be in locations where the user can find them easily.
   - Menus should not only be but also look interactive.
   - Menus should be emphasized and elevated from background clutter.
   - Menu link text should contrast with the background color of the menu.
2. Communicate the Current Location
   - Menus should mark the current location of the user.
3. Coordinate Menus with User Tasks
   - Menu labels should be easily understandable to the user.

   - Menu labels should be easy to scan for the user.
   - Large menus should preview lower-level content, to accelerate user navigation.
   - Menus should provide local navigation to closely related content.
   - Visual representations such as images or colors *can* aid user comprehension.
4. Make It Easy to Manipulate
   - Menu items should have an appropriate size and be easily clickable.
   - Previews of lower-level content, such as drop-downs should have an appropriate size.
   - Long application pages should contain menus that remain visible on top of the viewport at all times.
   - Most frequently used commands should have an easy access.

These guidelines are a good starting point to avoid common mistakes, nonetheless one has to consider that VR menus are presented in three dimensional space. In [32], Bowman writes that two dimensional interactions in a three dimensional world can be quite useful, yet if presence is important and immersion is to be preserved, then the menu has to be embedded into the virtual world and not just projected on screen space.

In [64], Bowman presents what he calls a distillate of *principles* of good 3D UI design:

1. Understand the Design Space
   Universal Interaction Tasks (see section 2.1.2) already exist. They can be reused directly or with slight adjustments in new applications.
2. There is still Room to Innovate
   Even though many techniques that have proven their worth already exist, it is still possible to establish new ideas and metaphors. The possible design space of 3D interaction is very large based on the multitude of input devices.
3. Be careful with Mappings and Degree Of Freedoms (DOFs)
   Incorrect mapping between input devices or DOFs (Fig. 2.3) of input devices and actions in the interface can lead to the confusion of the user. A wrong input device, such as for instance using isometric sensors like a SpaceBall (by 3Dconnexion) for position-controlled movements instead of isotonic sensors such as a position tracker, can decrease performance [73]. Interaction can also be made unnecessarily difficult if a high-DOF input is used for a task that requires a lower number of DOFs. For example selecting an item in a flat menu in a VE with a three dimensional input technique like a Pinch Glove, which would only need a two DOF device. Generally designers should try to reduce the number of DOFs with physical or virtual constraints.
4. Keep it Simple
   UIs in 3D can support complex tasks, yet not all tasks need to be made complex. Designers should user simple techniques for simple user goals. For example by reducing the number of DOFs as mentioned before.
5. Design for Hardware
   In flat menu design, the goal nowadays is to make every interface work on every device. Websites for instance should be displayed correctly on desktop computers, tablets and smartphones alike. With 3D UIs however, what works on one input or

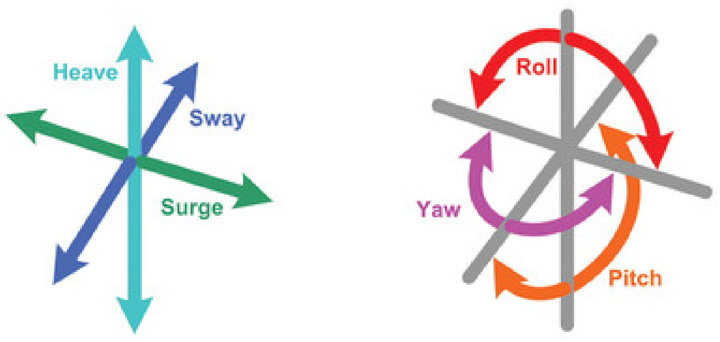


Figure 2.3: Degrees Of Freedom.

Translation		Rotation	
Surge	Moving forward/backward	Pitch	Tilting side to side
Sway	Moving left/right	Roll	Tilting forward/backward
Heave	Moving up/down	Yawn	Turning left/right

output device might not work on another. This migration issue can be demonstrated with the Arm HUD presented in section 2.2.1. The interaction works perfectly fine with a Leap Motion controller, however trying to get it to work with just a Wand is impossible.

6. Instruct Users
   One might think that interaction in three dimensional space is easy due to the fact that we humans interact in it all the time in the real world. For most novice users however, interaction with three dimensional interfaces can be quite unnatural [31]. Short instructions can be very helpful and improve user performance significantly.
7. Always Evaluate
   Evaluations should be conducted in all HCIs. Since interactions in three dimensional space can be quite complex though, evaluation should be done early and often.

In conclusion, there are no universally accepted guidelines for *3D Menus* yet. The developers have to choose the right approach for a specific task and evaluate their choice thoroughly. The general guideline for all UI design is as always: it should be easy to use and easy to learn.

2.3 Interaction with a Smartwatch

Interaction with wearables, in particular smartwatches are described in this section. First interactions by touch, then with the help of a IMU sensor and finally other approaches are presented.

2.3.1 Touch-based

There are different forms of touchscreens, as described in [27]: *resistive* Liquid-crystal Display (LCD) touchscreens, *capacitive* touchscreens, *infrared* touchscreens and *surface acoustic wave* touchscreens.

Almost all smartwatches available now, including the Huawei Watch, the LG Watch Urbane, the Samsung Gear S and the Motorola Moto 360 use *capacitive* touchscreens. Such touchscreens possess an insulator layer that is coated with a material that acts as conductor and stores electrical charges. When the layer is pressed, circuits on the corner of the device measure the charge and send that information to the controller. Capacitive touchscreens are very precise and resistant to contaminants. Unfortunately they cannot be used with gloves or other insulating materials since they need an electrical conductor touching the surface.

Capacitive touchscreens are used for wearables and most smartphones nowadays, because they are more responsive than resistive touchscreens and more resistant to contaminants than infrared and surface acoustic wave touchscreens. Additionally, since capacitive screens only consist of one layer which gets thinner and thinner as technology advances, the display itself can be many times sharper than with other methods.

Although touchscreens are used in nearly every smart device, their biggest disadvantage is that the user's hand may obscure the screen. This is especially the case with devices that possess a small screen, such as for example smartwatches. That is the reason why different approaches to smartwatch interaction may provide certain benefits over touchscreens.

2.3.2 IMU-based

The second most used interaction with a smartwatch or smart devices in general, is with the help of the sensors of the IMU. An IMU is a single unit consisting of two or three separate measurement sensors. The first sensor is the accelerometer which measures acceleration along each of the three axes and generates an analogue signal for each one. The second sensor is the gyroscope which measures rotational attributes like pitch, roll and yaw of the unit. Sometimes IMUs also contain a third sensor, a magnetometer that measures strength and direction of magnetic fields to help calibration against orientation drift.

IMUs are now based on Micro-Electro-Mechanized-System (MEMS) technology to enable use in smart devices to track the user's movement. The sensory data they provide can be utilized for activity tracking [67, 48, 61], fitness tracking [41] or as remote input for other applications such as in [43] by Kim and Woo. They used IMU data of a Samsung Gear Live smartwatch fused with depth tracking data from a sensor mounted on an Oculus Rift DK2 as a six DOF hand movement tracker in an Augmented Reality (AR) environment. The authors compared their implementation with another hand tracking technique explained in [51] that uses only depth data. User tests showed that the new technique improved completion time of a user task by 3 times. In [42], *Watchpoint* is explained which is an interaction technique that uses only the IMU of a smartwatch

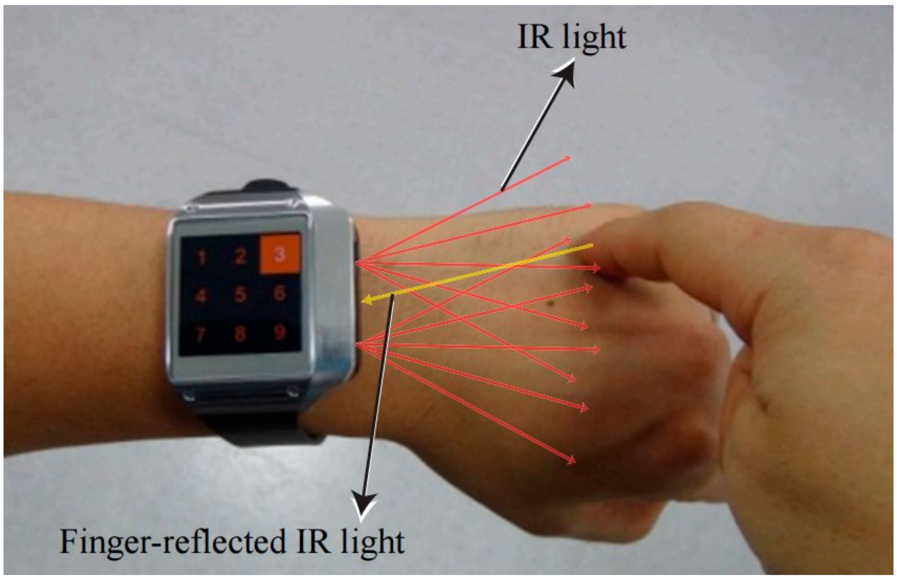


Figure 2.4: Prototype enabling Infrared (IR)-based interaction.

as tracking input. The authors use the sensors of the smartwatch to enable freehand mid-air pointing and clicking on a nearby large screen. The expected result was a natural and intuitive way to point on objects and select via custom gestures without the constraint of having to use a camera-based sensor. A study was conducted with a prototype showing promising results in such a way that the authors deemed *Watchpoint* a potential *killer-app*.

2.3.3 Other

There are many other techniques that have not yet been explored as much and are not used as a common interaction technique in nowadays smartwatches. All presented techniques in this section have the same fundamental idea: Avoid screen space touch interaction and as a result occlusion of displayed information.

In [46], the smartwatch touch interface is expanded around the screen to the skin of the user with the help of an IR line image sensor. While the hand is in an interaction position, which is verified with a gyroscope, the user can touch the back of his hand to control the graphical user interface of the smartwatch. The authors created a module that contains the line image sensor and two IR emitters that are mounted on a Samsung Galaxy Gear smartwatch; specifically on the side that is facing the hand. When a finger is anywhere on the back of the hand, the reflected IR light is detected and thus the position estimated (Fig. 2.4). To evaluate the performance the authors tested the technique with a six DOF industrial robot and found that the measured error rate amounted only to approximately seven millimeters in x and four millimeters in y direction. These results mean that the interface might not yet be used for sub-millimeter precise interaction but can definitely be used for scrolling and clicking interaction with the UI.

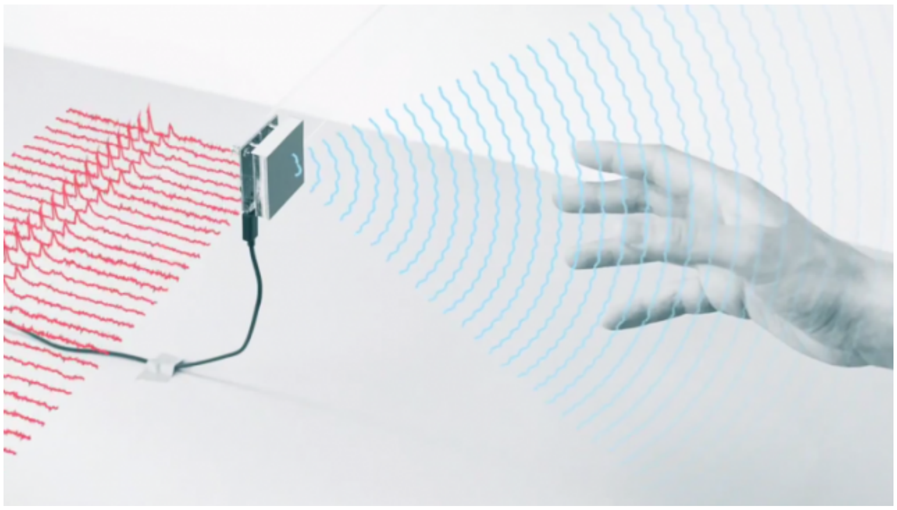


Figure 2.5: Project Soli by Google.

In [53], *ProxiWatch* is presented by Müller et al. They point out that touch-based interaction with a smartwatch can lead to occlusions of the displayed information, since the average screensize of the device is only about 4 centimetres in diameter and the trend is going torwards even smaller devices [54]. This is why they propose to use the additional DOF provided by the elbow joint which can flex to and extend away from the body of the user. This way users can for instance perform scroll interactions with only one hand by moving it from or away from their bodies. To achieve this, the authors constructed a stand-alone wireless prototype and mounted it on a Motorola Moto 360 smartwatch. The prototype consisted of a battery powered Arduino Nano and two IR distance sensors. The IMU of the smartwatch detects if the hand is raised; after that event, the IR sensors send the distance data via the Arduino wirelessly to a smartphone that processes the data and sends it back to the smartwatch for appropriate action. For evaluation, the estimated distance by the prototype and the real distance between smartwatch and body were measured and deemed as robust with a deviation of about one to two centimetres.

Project Soli [16] was presented at this year's Google I/O developer conference by Google's Advanced Technology and Projects (ATAP) group. It is a new way for touchless interaction with smart devices. The sensor (Fig. 2.5) works with radar technology to track the user's hand and finger gestures up to a sub-millimeter level. The goal is to have the human hand as universal input device for interaction with smart devices and make interaction feel as physical and as responsive as possible by using the haptic sensation of the fingers touching each other. The sensor can, for example, determine differences between a sliding or pressing movement of two fingers up to 15 meters away from the device (depending on the size of the chip). At the conference the group showed a working prototype where the chip was implanted into the wristband of a smartwatch and promised to make developer kits available in Fall 2016.

A method for mechanical bezel interaction with a smartwatch is presented in [71].

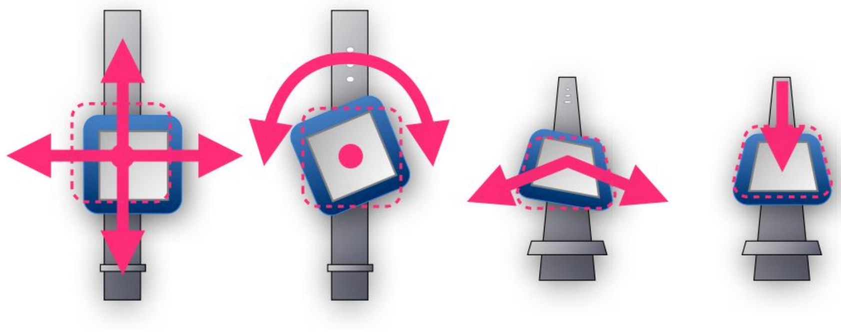


Figure 2.6: Mechanical interaction with a smartwatch.

The actions supported by the bezel are two dimensional panning and twisting, binary tilting left or right and clicking (Fig. 2.6). The prototype consisted of a small Thin-Film-Transistor (TFT) LCD color display mounted on top of a pair of joystick sensors to capture two dimensional movement. Overall the prototype worked well and provided new ways to interact with smartwatch content without having to occlude the screen by touching it. However the authors also point out that the problem with any form of mechanical parts on wearable devices is that they are less resistant to contaminants and prone to be affected by water entering the openings of the parts.

To conclude, smartwatches are not particularly new, nevertheless they are still a market niche. The first Android Wear devices launched in June 2014, but only really decoupled from Android Smartphones and exist as a solo device since Version 1.3 which was released much later. Now, in 2016 approximately 50 million people own a Smartwatch [24] which is nowhere near the 2400 million people that own Smartphones [23]. Interaction exploration with such small wearable devices is still only on the brink.

CHAPTER 3

Design

This chapter presents the design ideas for the two proposed sub-systems — the Android applications and the UE4 plugin — as well as the communication between them. First background information on languages, platforms and tools of each system are given. After that, the requirements and the designing ideas are presented. Lastly, the communication between the Wear and the UE4 system is discussed.

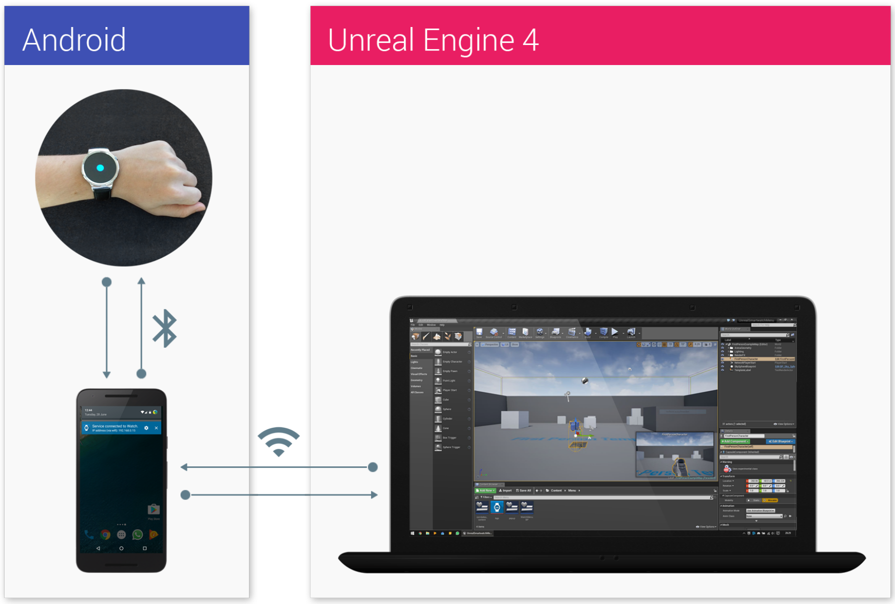


Figure 3.1: Schematic of the Wear-UE4 system. The Android Wear application connects to the Android smartphone application via Bluetooth. The smartphone application connects to the UE4 plugin via a WLAN.

Code Name	KitKat	Lollipop	Marshmallow	Nougat
Version Number	4.4 - 4.4.4	5.0 - 5.1.1	6.0 - 6.0.1	7.0
API Level	19 - 20	21 - 22	23	24
Wear API Level	1.0	1.3	1.5	2.0

Table 3.1: Android Versions.

3.1 Android Applications

3.1.1 Language

Java [38] is an object-oriented, class-based, platform-independent computer programming language developed by the Oracle Corporation. Java is arguably the most popular programming language.

One of the main design goals of Java is portability. Programs that are written in Java have to be able to run on different hardware and software. This is achieved by compiling the code into specific Java bytecode instead of architecture-specific machine code. This bytecode can then be executed on every hardware by the corresponding java Virtual Machine (VM) which translates the bytecode into machine-specific code.

Another main idea behind Java is simplification of memory management for developers. As opposed to other languages such as for example C or C++, the length of the lifecycle of an object is determined by the runtime environment and not by the programmer.

3.1.2 Platform

Android [1] is an open-source mobile Operating System (OS) based on the Linux kernel developed by Google. The UI of Android is based on direct touch manipulation of on-screen objects. The OS is available for smartphones, tablets, smartwatches, televisions and car dashboard units, each using the Android Application Programming Interfaces (APIs) and having a unified UI experience. In table 3.1, Android versions since the release of Android Wear are displayed.

Applications for Android are written in Java which provides the platform with multiple advantages including the following most important ones:

1. Java runs in a VM so there is no need to recompile an application on every different smartphone. The VM also has the advantage of separating processes, thereby making it more difficult for rogue applications to manipulate or interfere with other applications.
2. Java is easier to program due to the easier memory management and the missing pointer arithmetic. Instead of pointers, Java uses *References* which are type-safe,

thus making it impossible to reinterpret bytes in memory. This can be problematic if a wrong or unused memory address is pointed to, after reinterpretation.

3. Java is the most popular programming language, making the developer base and thus the number of available applications enormous.

3.1.2.1 Activities and Lifecycle

In Android, activities are the foundation upon which screens for an application are built. An application usually has multiple activities, each creating and handling different screens. In Fig. 3.2 the lifecycle of an activity is displayed.

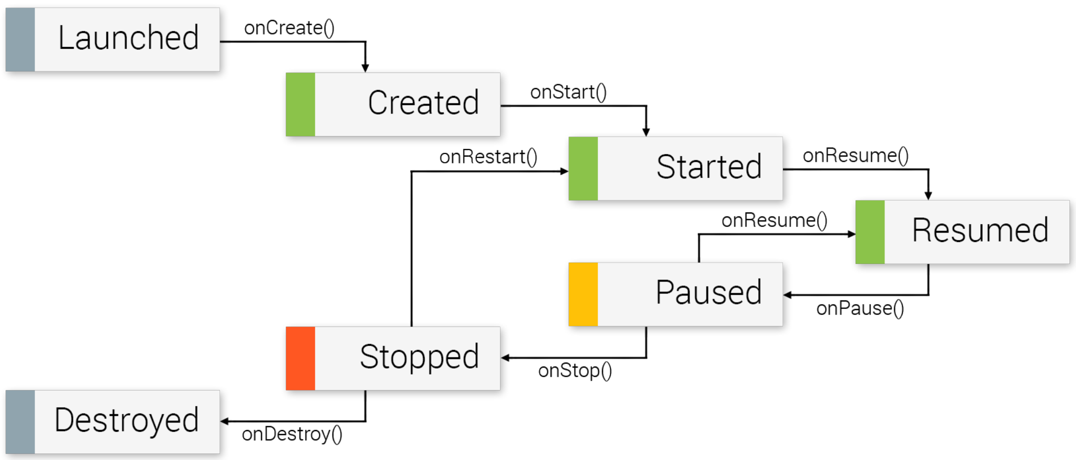
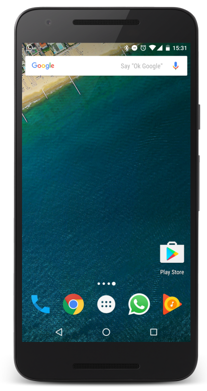


Figure 3.2: Lifecycle of an Android Activity.

Handling lifecycle stages and knowing when an activity is in which stage, is crucial when implementing a reliable application. When an application is launched by the user, the main activity is created which automatically calls the *onCreate*() method. In this method, UI elements and data objects are initialized. After the *onStart*() and *onResume*() methods are called, the activity is brought to the foreground. In this state the user perceives the application as opened. When the activity slides into the background, for example when the user navigates to the home screen of the device, the *onPause*() and the *onStop*() methods are called. In this state, the activity is still alive in the background; however as soon as the activity is destroyed by either the system or the user, the *onDestroy*() method is called.

The UI of an activity is defined in a layout which declares the different interface elements as visual structure. Layouts can be created either during runtime by creating *Views* or in a separate Extensible Markup Language (XML) file. Android has its own XML vocabulary to create the layout file, however most modern Integrated Development Environments (IDEs) have a graphical editor that, after designing, generates the corresponding XML layout file automatically.


(a) LG Nexus 5X

(b) Huawei Watch

Figure 3.3: Hardware.

Android applications can also be created without the need of an UI thread, in other words run in the background as a service. This allows to run more time consuming operations without affecting the interface's responsiveness. This way users can open and interact with other applications while the service is still operating unseen.

3.1.3 Tools

This section describes the different hardware and software tools that are used to implement the Android applications.

- Hardware
  1. LG Nexus 5X (Fig. 3.3a)
     An Android smartphone manufactured by LG Electronics. It has a ARMv8-A processor, 2 Gigabyte (GB) Random-Access Memory (RAM) and is running Android version 6.0.1 *Marshmallow*.
  2. Huawei Watch (Fig. 3.3b)
     An Android Wear-based smartwatch manufactured by Huawei. It has a Qualcomm Snapdragon 400 processor, 512 Megabyte (MB) RAM and is running Android Wear version 1.5.
- Software
  1. Android Studio
     Android Studio [2] is used for the implementation of the Android Wear application as well as the Android background service on the smartphone. It is the official IDE by Google for Android platform development. It is freely available and based on IntelliJ IDEA which is a Java IDE. Features include Gradle-based build support, Lint tools for compatibility evaluation, debugging support for all Android devices as well as emulated devices and a rich UI layout editor with support for every type of device.

3.1.4 Communication between Phone and Watch

Sending data between an Android smartphone and a paired Android Wear device is possible with two different APIs provided by the Android framework: The *DataAPI* and *MessageAPI.*

The *DataAPI* provides a communication channel for long-term synchronizing data between the devices, or nodes, in a network even when some devices are not online. The data is synchronized eventually when the node comes online.

The *MessageAPI* on the other hand is used for short-term messaging between connected nodes. A message is only considered as successfully delivered if it is queued to be sent to the receiving node. The receiving node must be online for this operation.

Since the communication between the smartwatch and the smartphone is only an exchange with very short messages (see section 4.1) and the nodes are always online and connected, the *MessageAPI* is used for the implementation.

3.1.5 Smartwatch Application

3.1.5.1 Requirements

In this section, the requirements for the smartwatch application are explained.

Interaction
The most important requirement concerns user interaction with the application. Due to the fact that users are wearing HMDs while they control the VR menus, the interaction with the smartwatch has to be as simple as possible. Also menus in UE4 are designed using UMGs (see section 3.2.2.1), with which two dimensional *Flat Menus* are created. Considering these two factors, the user needs to be able to input commands to two dimensionally navigate through the menu and also choose a desired menu item. Since the application could be used in a VE where the user is fully body tracked, a switch between interaction mode and idle mode of the VR menu has to be made distinctly via a gesture or a different input pattern. The VR menu is not supposed to be activated without the user specifically wanting it. Additionally the user has to be able to input text commands to fill out text fields in the VR menu.

Haptic Feedback
Since users are wearing HMDs, they do not need any visual feedback on the screen of the smartwatch. However the developer of the VR menu still has to have the ability to provide the user with haptic feedback to indicate special events such as for example that the device now listens for speech input.

Independence.
The application has to be independent. The navigation of the VR menu needs to work without any additional data input from another system such as for example any body tracking data that the VR system could provide.

Compatibility
The application has to be compatible with multiple Android devices. As minimum Software Development Kit (SDK), API level 21 is to be chosen. This version is available for nearly all Android smartwatches that are available right now. Additionally, to be able to support as many devices as possible, and also for battery consumption reasons which are discussed later in this section, the input data that is sent to the UE4 plugin over a WLAN has to be sent to the smartphone application first. This is mainly because not all smartwatches have the ability to connect to a WLAN by themselves and it is assumed that everyone who owns a smartwatch also owns a companion smartphone. Another reason why the smartphone is required, is that there is no possibility to input text via a software keyboard on devices that run Android Wear version 1.5. In 1.5 only speech input is available, as opposed to Android Wear version 2.0 where the Input Method Framework (IMF) is extended with a software keyboard and handwriting input. This version is however not available at the point of writing this thesis.

Resource Usage
The Huawei Watch has 4 GB of internal storage and no possibility of extending it. This is the standard size for storage on smartwatches right now, with watches such as for example the LG Watch Urbane, the Samsung Gear S or the Motorola Moto 360 having the same memory size. Since users can store media such as songs on their smartwatch locally to be able to play it without their phone nearby, storage space is quite precious. For that reason the size of the application needs to be held as small as possible.

Also Central Processing Unit (CPU) usage has to be as minimal as possible. Smartwatches have relatively small Li-Ion batteries which range between approximately 300 and 400 Milliampere-Hour (mAh). Since sessions in VR can take up to multiple hours and the watch is probably used or at least the application is opened all the time, battery consumption via intense CPU usage has to be prevented. This is the second reason why the smartphone companion application is implemented as well. The *heavier lifting* has to be done by the smartphone which has a larger battery.

3.1.5.2 Design Ideas

In this section, the general design ideas on the basis of the previously defined requirements for the smartwatch application are presented.

Interaction
To achieve the requirement of simple two dimensional navigation, two interaction options have to be considered. Touch-based and IMU-based methods which were discussed in section 2.3, are the only ones possible for consumer smartwatches at the moment of this thesis. IMU-based interaction via gestures or pointing however have some disadvantages, such as for example the drift of the sensor. To counteract the drift of the IMU, some form of absolute position data would have to be used. This data might be available, depending on the VR system, however to attain the requirement of independency from other systems, the application should not use this data. Another disadvantage of IMU-based interaction

might occur due to the constant movement of the user in the VE. Users might be fully body tracked in the VE, thus very unnatural or complex gestures might have to be performed to toggle the state of the menu and to keep it from activating unintentionally.

Therefore touch-based interaction with the smartwatch application is selected for two dimensional navigation. Input has to happen via swiping on the screen in the direction that the user wants to navigate. Furthermore users have to be able to continuously swipe in a direction to scroll over multiple menu items, depending on how long they swipe. The length in pixel of one swipe has to be customizable however since the screen of the smartwatch is rather small, this customization has to be performed on the smartphone. Clicking has to be performed with simple tapping onto the screen of the device. To switch between input mode and idle mode of the VR menu, users have to perform a distinct touch gesture that they would normally not perform, to prevent unintended activation of the VR menu. For this reason, a double tap is chosen as activation and deactivation touch pattern. What also has to be considered, is that in Android Wear 1.5 swiping the current application to the right of the screen, dismisses the application. This has to be disabled during the start of the application to enable touch-input for the VR menu in all directions. However if this gesture is disabled, there needs to be another possibility to close the application. To exit *Full-Screen Activities*, Google suggests a long press to dismiss pattern [5] that overlays the current Activity with a *Quit* button which can be utilized by the user.

Since input via a software keyboard is not possible in Android Wear 1.5 and the user is wearing a HMD and should not have to take it off for any menu interaction, speech input has to be used for text entries. As soon as the user clicks on a text box in the VR menu, the smartwatch has to switch into speech-input mode. As soon as the user is done with the voice input, the smartwatch has to switch back to touch-input mode, translate the recorded speech and send the translated text to the smartphone application which then sends it to the UE4 plugin.

Haptic Feedback
Vibration is the only possible way of giving any feedback to the user via the smartwatch. The decision when haptic feedback is given to the users, is not to be made by the application itself. The decision has to be given to the developer of the VR menu, to for example vibrate the watch when the user selects a button or a text field. The developer also has to be able to decide how long the vibration lasts by sending a command with a time variable from the UE4 plugin.

Compatibility
Android smartwatches exist with square, round and round with chin on the bottom formats and different screen resolutions. To enable the compatibility with multiple devices, the application layout needs to be designed relatively to the screen size of the smartwatch. However this is mainly important for the *Quit* button, since there is no other screen position specific interaction with the application. Swipe and tap actions to navigate the VR menu can be performed anywhere on the screen of the watch.

Resource Usage
To address the minimization of resource usage, battery consumption is to be kept to a minimum. Screen are in general the biggest factor of battery drainage on any smart device. Due to the fact that the Huawei Watch has an Active-Matrix Organic Light-Emitting Diode (AMOLED) screen, it is possible to save a lot of battery by blackening the UI. Normal LCD screens have a backlight which is simply blocked when a black pixel occurs. However it is still on and consumes battery. AMOLED screens work without backlights; instead every pixel has its own light. That way, the screen does not need to block any light for a black pixel, it simply does not light up, thus needing no energy for this pixel.

To further reduce resource usage, the sending and receiving of data via the WLAN is relocated to the smartphone application.

3.1.6 Smartphone Application

3.1.6.1 Requirements

Here, the requirements for the smartphone application are explained.

Simplicity
The topmost requirement, same as for the other part of the Android system, is simplicity. Users have to not be obliged to interact with the smartphone application every time they want to navigate in a VR menu. This means that the smartphone application has to start and stop as soon as the smartwatch application is started or stopped. In addition to that, the smartphone application is not to be closed by unintended touchscreen- or other interaction. Consequently this means that the smartphone application has to start as a background service without main activity.

Layout
Despite the smartphone application being without a main activity, it still has to notify the user that the service that sends and receives data from the WLAN is running in the background. Furthermore it needs to be possible for users to adjust certain settings such as network options and the option to change the distance of a swipe, as explained in section 3.1.5.2. The settings have to be displayed in an activity that is only shown when the user specifically wants it, to prevent cluttering of the screen and UI responsiveness.

Compatibility
The smartphone application also needs to be compatible with multiple devices. Same as with the smartwatch application, the minimum SDK has to be API level 21. With this level, approximately 45% of all devices running Android [3] and 100% of devices that are able to connect to a smartwatch are supported.

Resource Usage
Requirements concerning resource usage on the smartphone are not as pressing as with

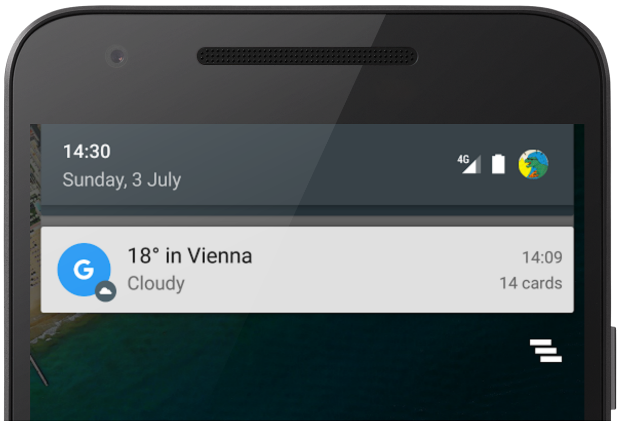


Figure 3.4: A notification in Android 5.0 - 6.0.1.

the smartwatch since phones have significantly larger batteries and storage. However CPU usage and thus battery drainage has to still be kept to a minimum due to the fact that the communication with the UE4 plugin does not work without the phone being turned on.

3.1.6.2 Design Ideas

Now, the general design ideas for the smartphone application are presented.

Simplicity
To achieve synchronous starting and stopping of the smartwatch and smartphone application, messages have to be sent from the smartwatch to the smartphone in the *onCreate*() and *onStop*() methods. For this, specific *Action Commands* (see section 4.1) are used which are sent to the smartphone with the help of *Google's MessageAPI*.

Layout
Since the smartphone application is to be run in the background but also provide a way to notify the user that the service is running, a notification has to be displayed in the notification bar (Fig. 3.4). This notification only has to be displayed when the service is running and provide the users with crucial information if needed. It has to indicate the state of the connection of the smartphone with the smartwatch and with the WLAN. Furthermore it has to provide the current Internet Protocol (IP) address of the phone combined with the used port which are needed to connect with the UE4 plugin and are further explained in section 3.3.

The notification has to include a button which when pressed, directs the user to the settings activity. The activity has to include three text fields that let the user set the client device port, the server device port and the IP address of the server device which are needed for the communication with the UE4 plugin. The settings UI has to include a

SeekBar which lets the user decide how long a swipe on the smartwatch has to be, to be detected as one scroll. The value of the SeekBar has to range between 20 and 200 pixels. Values beneath 20 pixels make scrolling too inaccurate on the smartwatch screen; values above 200 pixels make scrolling feel too slow. Furthermore the activity has to include a *Save* button to update the changes and a *Reset* button to reset the changes. Before any values are saved, they have to be validated. Port numbers are saved as unsigned 16-bit integers, so their values are restricted to $0 < port < 2^{16} - 1$. An example for a valid IPv4 address would be 192.168.0.3. If the new values are not valid, users have to be informed about that with a small popup.

Compatibility
To support multiple devices, the layouts for the notification and the settings activity have to be designed relatively. Furthermore German and English language support have to be provided by extracting the UI strings and keep them in an external file. These different languages files are kept in the resource directory of the Android project and are automatically applied, when the language of the device is changed.

3.2 Unreal Engine 4 Plugin

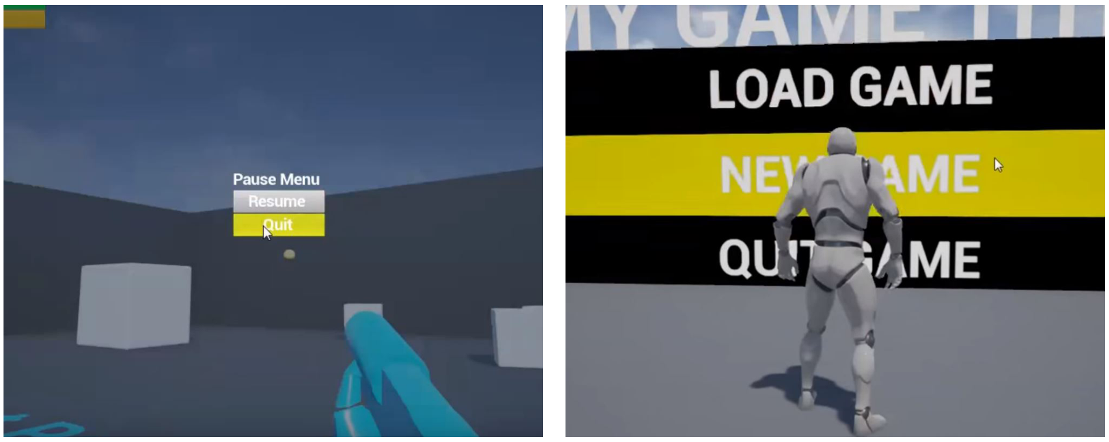
3.2.1 Language

C++ [66] is an imperative, generic, object-oriented programming language. It was released as an iteration to C (hence ++), with such advantages as for example virtual functions, operator overloading, inheritance and abstract classes; many of which are now standard in modern languages. C++ is useful in many different contexts such as in desktop applications, servers and performance-critical applications. It is a compiled language, same as Java, and not bound to a specific providing organization. A difference compared to Java is that every coding class has a *.h* (header) file and a *.cpp* file. The header file declares *what* is being implemented and the .cpp file declares how it is implemented.

3.2.2 Platform

Unreal Engine is a game engine developed by Epic Games, originally with the focus on First-Person Shooter (FPS) games; but is now used in many different genres such as RPGs, Multiplayer Online Battle Arena (MOBA) games and Massively Multiplayer Online (MMO) games.

The current version, UE4, is designed for DirectX 11/12 on Windows, OpenGL on Mac and Vulkan on Android and free for all platforms. UE4 supports, in contrast to previous versions, game scripting in C++ and visual scripting in the editor in so called *Blueprints*. Blueprints are special assets that are used to create actors, playable game pawns, script events, Head-up-Displays (HUDs), prefabs, etc. Depending on which parent class is chosen, every blueprint has different options and panels. However each blueprint provides an *Event Graph* which is a graphical interface that looks similar to a flowchart. There



(a) A menu in screen space. (b) A menu in world space.

Figure 3.5: UMG in UE4.

the developer can place nodes, events, functions or variables and connect them with wires to create complex procedures. Instead of textually specifying code, the developer is able to create and manipulate programs graphically. This a huge advantage for game designers that do not know how or do not want to program in the classical sense. The newest version also drastically reduces build time by making it possible to compile C++ code while the engine is running, making it overall easier to debug.

3.2.2.1 Unreal Motion Graphics

With the UMG UI designer tool the developer can create UI elements in the UE4 editor in a graphical interface. These elements can then be placed onto screen space or into world space. Screen space is a two dimensional bounded space defined in pixels by the screen (Fig. 3.5a). World space on the other hand is a three dimensional endless space in which camera and game objects are placed (Fig. 3.5b).

UMG is essentially a blueprint extension based on Slate. Slate is a custom UI window programming framework, designed originally to create the interface of the UE4 editor. It can also be used for interfaces inside a game, however it is based on a textual declarative syntax to compose UI elements. To make it accessible to more developers, the UMG wrapper was implemented to make the creation of interfaces possible in blueprints, therefore graphically.

To create a menu interface, a widget blueprint has to be created in which UMG widgets can be constructed with different components and functionality can be added. Widget components include interactive components such as buttons, check boxes, sliders but also panel components which are used to create a visual layout and define position of other widgets such as horizontal and vertical boxes, grid panels, scale boxes or widget switchers. The widget components are placed and arranged inside the designer tab which presents the developer with a graphical representation of the interface as well as a widget tree. This widget tree organizes the widgets in a tree structure where each widget has exactly one parent widget and zero or more child widgets.

Besides the designing tab, developers can also add functions to the widget with the help of visual scripting in a graph window. An example for a functionality of a widget could be a health bar that updates itself as the player fights with enemies and gets weaker.

3.2.2.2 Plugins in Unreal Engine 4

A plugin is a software module that adds features to an existing program. In UE4, many subsystems were designed to be extensible with plugins, allowing to add or modify functionality without altering engine code directly. It is possible to add new file types, new menu items or new tools for the editor.

Generally, two types of plugins exist in UE4: Engine Plugins and Game Plugins. *Engine Plugins* are plugins that are placed in the engine folder of the computer system and thus can be used across all projects that are implemented on that system. *Game Plugins* are plugins that are added to a single project thus can only be used in this specific project.

Both types of plugins can be either Content Modules or Code Modules. *Content Modules* contain content that is available in the game as well as for developers in the editor. An example would be a plugin that adds in-game character assets, animations or level components which developers can add to their own games. *Code Modules* contain code with which other modules can interact. An example would be a plugin that adds additional input possibilities to the editor, such as for example joystick input or input via a smartwatch.

Every plugin has to have an icon that is displayed in the editor's plugin UI. Furthermore every plugin has to have a module asset which describes the contents of the plugin such as name, category, description, version and engine version so that UE4 can display this information in the editor as soon as the plugin is loaded. Additionally it contains information on whether the plugin is an editor module, a developer module or can be used during runtime. Runtime plugins can be used in the shipped game, editor modules are only loaded when the editor is used and developer modules are loaded during builds of the game.

3.2.3 Tools

This section describes the different hardware and software tools that are used to implement the UE4 plugin.

- Hardware
  1. XMG Laptop
     For the development of the UE4 plugin, a laptop by XMG with an Intel Core i7 quad-core processor, 16 GB of RAM and a Nvidia GTX860M is used.
- Software

1. Microsoft Visual Studio 2015
   Visual Studio 2015 is used to implement the Unreal Engine 4 Plugin. It is an IDE that supports different programming languages and provides debugging support on source-level and a machine-level. Built-in languages include C, C++, C# and others.
2. Unreal Engine 4 Editor
   The UE4 Editor with version 4.12 is used to evaluate and test the UE4 Plugin and create multiple sample menus.

3.2.4 Requirements

In this section, the requirements for the UE4 plugin are explained. Since the design of the plugin will have an effect on both the developer and the user of the VR menu, the requirements are regarded respectively.

3.2.4.1 Users

Custom Components
Most importantly, the plugin needs to contain various possibilities which allow the user to input different information. These possibilities need to be in the form of custom widget components that can be traversed and selected via the smartwatch.

Visual Feedback
When navigating the menu, users needs to have feedback on where they currently are in the menu. Thus menu components need to be able to be highlighted as soon as they are scrolled over. Furthermore when operating special components such as sliders or combo boxes, users need to be informed when the element is focused so that they know when values can be edited via swipes.

Performance
In UE4 interfaces are usually controlled with a mouse or with a controller. However with the smartwatch plugin, the menu is navigated with the help of commands that are sent over a network and then processed. Despite that processing, the menu needs to operate swiftly and reliably which means that the algorithm has to be efficient and stable.

3.2.4.2 Developers

Plugin
First of all, the additional custom widget components need to be easy and fast to import into the UE4. Thus a runtime *Code Module Game Plugin* has to be created for the UE4 editor.

Custom Navigation
Furthermore users need to be able to navigate intuitively through the created menu, be

it with or without boundaries on the edges of the interface. Thus the menu navigation needs to be customizable by the developer to a point where a certain scroll command leads the user from one specific menu component to another specific menu component. However if the developer does not want to customize navigation, it also has to work without modifying it.

Scalability
The possible number of VR menus has to be scalable in regard to the number of users. Multiple users need to be able to simultaneously use a menu with which only they can interact.

Debugging
Another requirement concerns debug information of the received input data and the state of the menu. This is needed by the developer to know what exact data is received from the smartwatch. Since this input data could be annoying and disruptive for the user in the VE, the visibility of debug information has to be customizable.

Extensibility
Finally, the implementation of the plugin has to be extensible to a point where a another developer can simply add custom widget components to the plugin code without needing to change any previously implemented code. To achieve this, good documentation has to be written and the code has to be lowly cohesive.

3.2.5 Design Ideas

In this section, the general design ideas for the UE4 plugin are presented.

3.2.5.1 Users

Custom Components
To create custom menu components that can be navigated with input over a network, the components have to be derived from standard UMG widget component classes and then extended with custom code. The newly created widget components are defined in the following list:

- Watch Button
  A component that can be clicked via a tap on the smartwatch when the component is hovered. When the button is clicked, an event has to be fired which can be further customized by the developer to execute an action such as for example quitting the game.
- Watch Check Box
  A component that allows the user to make a binary choice between *Yes* and *No* by tapping the smartwatch when the component is hovered. When the state of the check box changes, an event has to be fired.
- Watch Combo Box
  A component that when hovered and clicked, opens up a drop-down list in which

the user can navigate to select an item. When the user taps the smartwatch again, the combo box closes. The box has to fire events on opening, closing and selection state change.
- Watch Slider
  A component that lets the user set a value by moving an indicator vertically or horizontally. When the slider is clicked, the user can control the slider handle with left and right swipes for horizontal sliders or up and down swipes for vertical sliders on the smartwatch. The slider has to fire an event when the value is changed.
- Watch Spin Box
  A component much like the slider, except that it also contains a number which displays the currently selected value.
- Watch Text Box
  A component that lets the user enter text. Since the user is not able to enter text on the smartwatch via touch, the text has to be supplied by speech input. When the box is clicked, the smartwatch switches to speech-input mode. When the input text is received in UE4, it is written in the text box and the box looses focus. If an error occurs, such as for example no internet connection which is needed for the speech service, then a debug error message is displayed and the component also loses focus.

Visual Feedback
Furthermore all components have to have the ability to indicate highlighting or focusing. For that, every widget has to have custom color edit fields that can be updated by the developer in the Class Default Object (CDO) panel. Some components already posses standard color fields, some need to be updated. Each component has to also fire an event when it is hovered or unhovered and focused or unfocused which can also be further customized by the developer of the VR menu.

3.2.5.2 Developers

Custom Navigation
To achieve the requirement of custom navigation, the developer has to be able to specify in each component, in which direction of the current component another component lies in the menu. Say for example that when a button is highlighted and the user swipes left on the smartwatch, the menu navigates to the component that the developer specified as being left of this button even though the component might not be physically placed left of the button. If custom navigation is not declared, then the next component in the component tree is chosen if the user swipes right or down and the last component is chosen when the user swipes left or up.

Scalability
To provide the possibility of multiple menus in one VE, every menu has to have the option to specify a network port to listen for input commands sent from a smartwatch.

3.3 Communication between Android and Unreal Engine 4

To communicate across a computer network, a socket is needed which is an abstract endpoint instance defined by an IP address, a port and a transport protocol. Two of the core transport layer protocols defined by the Internet Protocol Suite (IPS) are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP) (Fig. 3.6).

TCP is a connection oriented protocol that functions as a stream of data over an IP network from one node to another. All data packets that are sent are guaranteed to reach the destination in the correct order they were sent. TCP is generally slower but very reliable, with typical applications such as web browsing or Email traffic.

UDP is a connectionless protocol. The communication works with the help of datagram packets that only guarantee the integrity of themselves. The packets can reach the destination correctly, out of order or not at all. It is however more time efficient since there is no handshake needed between client and server. Typical applications include streaming media and real time communication such as Voice over IP (VoIP).

In the implemented system, the UE4 plugin will act as a server since it provides the service of menu navigation. The plugin gets input requests by the Android application which it responds with visual output in form of menu navigation.

The request and response messages that are sent between the system are comparably short commands but the server has to respond fast to the requests of the client to avoid input lag during fast input events. In addition, the system has to have an easy one-time setup process. It also has to be possible for users to simply connect to a certain menu while the VE is already in use. Thus a connection oriented approach is disregarded and the communication is implemented with the help of the UDP. As the distance between the devices is comparably small in the WLAN, it is quite unlikely that any datagram packages will be lost during the transmission, thus this problem of UDP communication is not an issue.

The cardinality of the Wear-UE4 system is defined as a one-to-one relationship. To communicate in both directions, each system has to have knowledge of the port that itself makes available for transmission. Additionally, the port and address of the respective counterpart has to be known. With this information, one Android application can only communicate with one VR menu.

Lastly, the possibility that no WLAN is available to transmit input data has to be considered as well. If the VR system does not provide a wireless network, the proposed system also has to work with a hotspot created by the Android smartphone. The device that is using the UE4 plugin connects to the hotspot. This circumstance however is mainly thought out for debugging purposes since most immersive VR systems depend on a WLAN to transmit sensory input data.

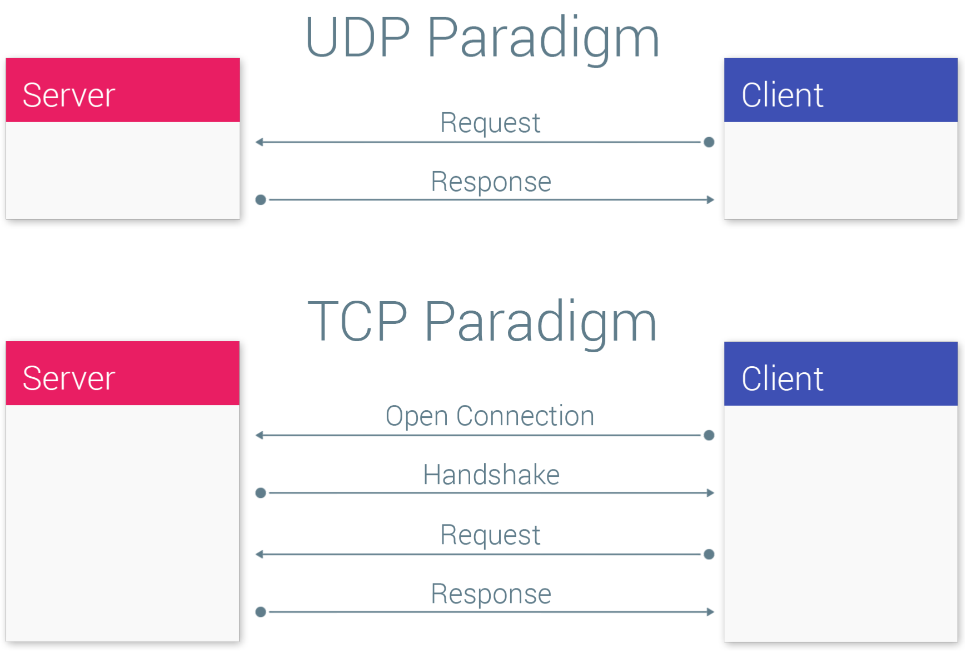


Figure 3.6: TCP and UDP communication comparison in a server-client model.

CHAPTER 4

Implementation

In this chapter, the implementation of the Wear-UE4 system is described. First the *Action Commands* that are used to communicate between all devices and then the implementation of the Android application and the UE4 plugin are presented. Finally the usage of the system is explained.

4.1 Action Commands

To enable proper communication between all three devices in the Wear-UE4 system, a consistent transmission code has to be established. Therefore specific *Action Commands* are created and presented in table 4.1. The table displays all available commands with the direction that they are sent, the string of the command, the possible attachment to the string for further information and an explanation of what the command does.

The commands have to be stored in a file that is available to every other class in the system. For the UE4 plugin, a separate class is created that contains each command as a static *FString*. FString is a specific string class used by UE4. Additionally a static method is implemented in the class to check whether a given FString is a viable *Action Command*: *static bool isViableActionCommand(FString Command);*

For the Android applications, a separate module is created that is imported into both platform implementations. This module contains a class with the commands as static strings and a similar static method as mentioned above.

In Fig. 4.1, an example for an exchange with the proposed commands is presented. First, the smartwatch application is started which sends the *ACTION_PHONE_START* command to start the smartphone application. As soon as the phone application is started, *ACTION_WATCH_SCROLL_DIST_50* and the *ACTION_WATCH_DEBUG_TRUE* are sent to the watch (see section 4.2.2.3). Afterwards, the menu is activated with the *ACTION_MENU_ACTIVATE* command. When the menu is opened, the user swipes down on the smartwatch and then *ACTION_MENU_SCROLL_DOWN* command is sent to

Direction	String	Attachment	Comment
Watch → Phone	ACTION_PHONE_START	None	Starts the smartphone application.
	ACTION_PHONE_STOP	None	Stops the smartphone application.
Phone → Watch	ACTION_WATCH_SCROLL_DIST_	Integer	Sets the minimal distance in pixels that is needed to perform a scroll on the smartwatch.
	ACTION_WATCH_DEBUG_	TRUE/FALSE	Sets whether visual debug information on the smartwatch is visible or not.
W → P → UE4	ACTION_MENU_ACTIVATE	None	Activates the VR menu.
	ACTION_MENU_DEACTIVATE	None	Deactivates the VR menu.
	ACTION_MENU_SCROLL_UP	None	Scrolls one item up in the VR menu.
	ACTION_MENU_SCROLL_DOWN	None	Scrolls one item down in the VR menu.
	ACTION_MENU_SCROLL_LEFT	None	Scrolls one item left in the VR menu.
	ACTION_MENU_SCROLL_RIGHT	None	Scrolls one item right in the VR menu.
	ACTION_MENU_CLICK	None	Clicks the currently selected item in the VR menu.
	ACTION_VOICE_RESULT_	String	Sends the speech result to the VR menu.
UE4 → P → W	ACTION_MENU_ACTIVE_	TRUE/FALSE	Informs the smartwatch whether the VR menu is active or not.
	ACTION_VOICE_START	None	Starts recording speech input on the smartwatch.
	ACTION_WATCH_VIBRATE_	Float	Starts vibrating the smartwatch for a given amount of time.
	ACTION_ANDROID_STOP	None	Stops the smartphone and the smartwatch applications.

Table 4.1: The available *Action Commands* across all three devices.

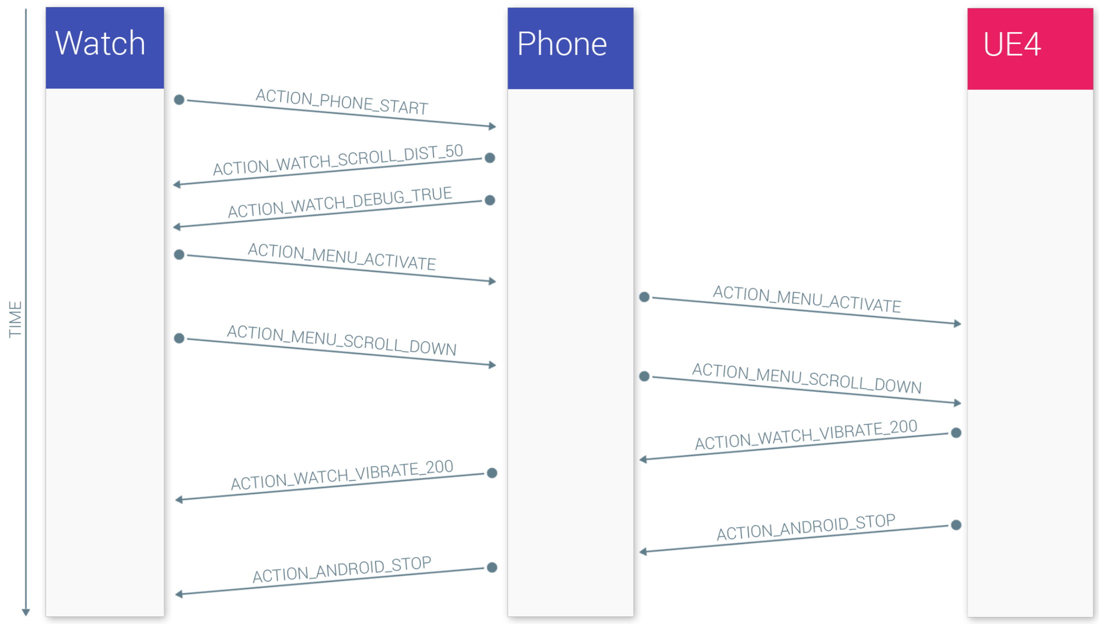


Figure 4.1: Diagram of an exchange with the help of *Action Commands*.

the UE4 plugin. After a time, the plugin sends the *ACTION_WATCH_VIBRATE_200* command which vibrates the watch for 200 milliseconds. Finally, the *ACTION_ANDROID_STOP* command is called from the UE4 plugin that stops the phone and the watch application.

4.2 Android Application

In Android Studio it is possible to create one project that contains different form factors. The project that is created for this thesis has the name *WatchPlugin*, the domain is *ims.tuwien.ac.at* and the application that runs on the wearable and the phone is called *Unreal Smartwatch Menu (USM)*. The minimum API level of the application is 21, whereas the target is 23. The application icon is shown in Fig. 4.2.

Figure 4.2: Android Application Icon

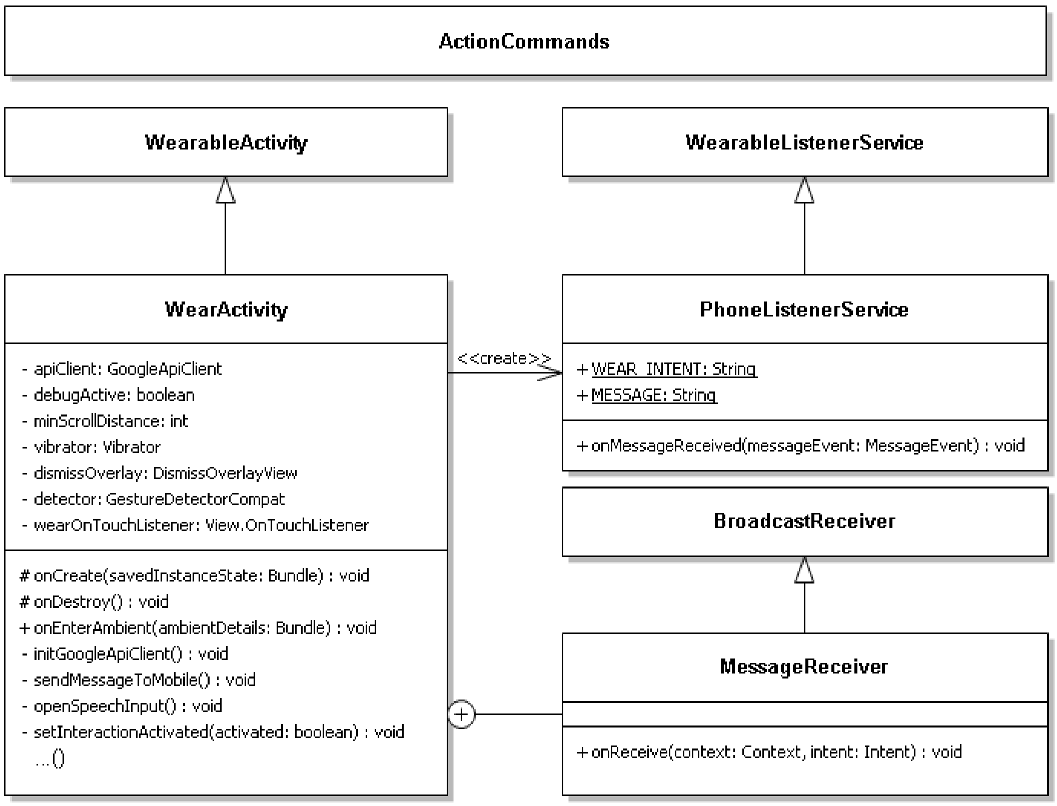

ActionCommands
WearableActivity
WearableListenerService
WearActivity
- apiClient: GoogleApiClient
- debugActive: boolean
- minScrollDistance: int
- vibrator: Vibrator
- dismissOverlay: DismissOverlayView
- detector: GestureDetectorCompat
- wearOnTouchListener: View.OnTouchListener
# onCreate(savedInstanceState: Bundle) : void
# onDestroy() : void
+ onEnterAmbient(ambientDetails: Bundle) : void
- initGoogleApiClient() : void
- sendMessageToMobile() : void
- openSpeechInput() : void
- setInteractionActivated(activated: boolean) : void
...()
<<create>>
PhoneListenerService
+ WEAR_INTENT: String
+ MESSAGE: String
+ onMessageReceived(messageEvent: MessageEvent) : void
BroadcastReceiver
MessageReceiver
+ onReceive(context: Context, intent: Intent) : void

Figure 4.3: UML Class Diagram of the Watch Application with the most important class variables and methods.

4.2.1 Smartwatch Application

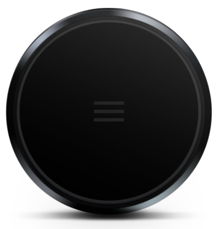
In this section the implementation of the smartwatch application is presented. The application consists of one activity called the `WearActvity` and a service called `PhoneListenerService`, as seen in the Unified Modeling Language (UML) class diagram in Fig. 4.3. The `WearActvity` is started as soon as the user opens the application.

First the required permissions, then the creation of the layout and after that the functionality of the application is explained.

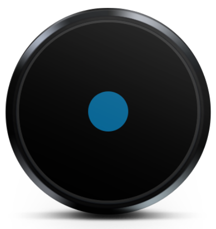
4.2.1.1 Permissions

In order to allow the watch application to vibrate the device and to keep it from going into sleep mode, the following permissions have to be added to the `AndroidManifest`.

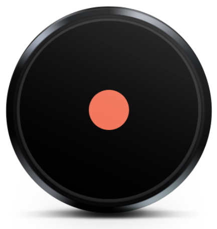
```
<uses-permission android:name="android.permission.VIBRATE"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
```



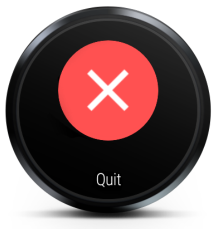
(a)



(b)



(c)



(d)

Figure 4.4: Android Wear Application UI.

The manifest is a file that presents essential information about the application that is needed by the OS. It contains the name of the java package of the application, shortly describes all activities, services, broadcast receivers and content providers of the application and declares the permissions.

4.2.1.2 Layout

Since Android Wear smartwatches exist with round, round with chin and square shapes, the layout of the `WearActvity` needs to be shape-aware. For that, a `BoxInsetLayout` which is included in the Wearable UI library is used. It lets the developer define a single layout that works with different screen types.

The main issue with supporting different screens, is that objects in the corners of square devices, might get cut off on round devices. To prevent this, child views are positioned relatively inside a `BoxInsetLayout` with the help of a parameter to align the view either on the top, bottom, left or right edge, or in the center of the layout. For the `WearActvity` layout, all views inside the `BoxInsetLayout` are centred. Furthermore the background color of the layout is set to black to reduce battery drainage.

Inside the `BoxInsetLayout`, a faded `ImageView` is placed in the middle of the screen to indicate that the application is indeed running (Fig. 4.4a). If visual debugging is enabled, a colored custom created `CircleView` is drawn on top of the faded image. This circle is only visible, when the VR menu is activated. The different colourings of the circle (Fig. 4.4b and Fig. 4.4c) are explained in the next section.

To be able to close the application, a DismissOverlayView has to be added to the layout, so that it is drawn full-screen and over all other views (Fig. 4.4d). To detect if the view is to be opened or closed, a GestureDetector is used which is further discussed in the next section.

4.2.1.3 Functionality

After the user opens the application, the *onCreate()* method is called, as discussed in section 3.1.2.1. Here, the layout specified in the resource XML is set as the content view of the activity. After that, the *setAmbientEnabled()* method is called which is provided by the WearableActivity parent class of the WearActivity. It makes sure that the application remains displayed even during ambient mode. Ambient mode is a way of the OS to automatically save battery life by closing open applications and dimming the display of the watch after a certain time of inactivity by the user.

Even though the application is not closed in ambient mode, the touch event recognition still goes into idle mode. To exit this idle mode, a touch event on the screen is normally required. To prevent the application from going into idle mode altogether, a wake lock is used in the *onEnterAmbient()* method. A wake lock is a mechanism to indicate that the device needs to stay powered up.

After that, the different communication channels are set up. First the *GoogleApiClient* is built for communication with the phone. Here, the phone's node Identifier (ID) is detected and saved for sending messages to the smartphone with the help of the *sendMessageToMobile(String message)* method. After that, the PhoneListenerService and the MessageReceiver are created. The PhoneListenerService is responsible for listening to incoming *MessageEvents* from the paired smartphone. When such an event is received and identified as viable *Action Command*, the service broadcasts that event with the help of a LocalBroadcastManager with a specific intent. An intent is a messaging object in Android used to request an action from a different application component such as services, activities, etc. In this case, the the broadcast is received by the MessageReceiver that is an inner class of the WearableActivity. This inner class is in charge of translating incoming *Action Commands* into actions on the watch such as for example vibration, setting the *minScrollDistance* or opening the speech input.

As soon as the communication channels are set up, the application sends the *Action Command* to start the smartphone service.

Finally, the handling of touch events is implemented in the *onCreate()* method. First, a GestureDetector is applied to the whole layout which detects various gestures and events, using MotionEvents provided by the OS. The detector provides a OnGestureListener callback which notifies users of the detector when a particular motion occurs. The following events are requested to be detected by the detector:

- *onLongPress(MotionEvent e)*
  On a long press, the DismissOverlayView is displayed. Since it is a view provided by the OS, MotionEvent and visibility handling inside is performed automatically.

- *onSingleTapConfirmed(MotionEvent e)*
  This callback is executed when a single-tap on the screen occurs. The application then sends a click *Action Command*.
- *onDoubleTap(MotionEvent e)*
  After a double tap on the screen, the watch either sends an activate or deactivate *Action Command* depending on the VR menu state. Additionally the `CircleView` is set to visible or invisible with the help of the *setInteractionActive()* if visual debugging is enabled.
- *onScroll(MotionEvent e1, MotionEvent e2, float distanceX, float distanceY)*
  This callback is executed when a scroll occurs. The method provides the initial `MotionEvent` that occurs when the finger is put on the screen and the `MotionEvent` that occurs currently, as the finger is moving. A scroll *Action Command* is sent as soon as the *minScrollDistance* is reached in a certain direction.

Additionally to the `GestureDetector`, an `OnTouchListener` is applied to the layout that provides a callback as soon as the screen is touched to support visual debug output. When the screen finger is put on the screen, the color of the `CircleView` is changed to red. When the finger lifts up, the color is changed back to blue.

When speech input is required, a new activity is started with a *RecognizerIntent*. The Google Voice Search application then responds to this intent by displaying a "Speak now" dialogue and streams any speech to the Google servers for further interpretation. This is the reason, why speech input requires an active internet connection. After the speech is translated to text, the dialogue is closed and the text is sent to the phone as attachment to an *Action Command*. During the input dialogue, it is possible to swipe right to dismiss the dialogue and cancel voice input.

When the application is closed by the user, the *Action Command* to close the smartphone service is sent and finally, all communication channels are closed.

4.2.1.4 Debugging Wear Applications

To debug a phone application, the smartphone has to be connected to the working station via Universal Serial Bus (USB) and the debugging option has to be enabled in the developer options of the phone. After that, log outputs can be displayed in the Android Debug Bridge (ADB) command line tool. The ADB lets developers communicate with emulator instances or connected Android devices.

In Android Studio, the developer does not need to debug via the ADB directly, since the IDE runs all commands automatically. The developer only needs to connect the device, press the *Run* button and the application is installed and ready for debugging.

To install the debug Android Application Package (APK) of an Android Wear application however, the watch needs to either have a form of USB port to connect to the development computer or has to be connected to the companion phone via bluetooth which is then connected to the computer via USB. With the latter, the debug output is routed from the watch to the phone with the help of the ADB. Since the Huawei Watch does not have a cable port, the debugging process is performed over bluetooth. To do that, bluetooth debugging has to be enabled on the watch and smartwatch debugging has to be enabled in the official *Android Wear Companion Application*. Finally, the commands shown in listing 4.1 have to be entered into the ADB to foward debug output to the phone. After that, the application can be run and debugged like a phone application.

```
adb forward tcp:4444 localabstract:/adb-hub
adb connect localhost:4444
```

Listing 4.1: ADB commands with a desired port.

4.2.2 Smartphone Application

In this section the implementation of the smartphone application is presented. The application starts with the `MainActivity` which starts the `ListenerService` and closes itself as soon as the service is opened. In Fig. 4.5 the UML class diagram of the application is displayed.

First the required permissions, then the creation of the layout of the notification and the `SettingsActivity` and after that the functionality of the application is explained.

4.2.2.1 Permissions

The following permissions have to be added to the `AndroidManifest` for the application to be able to send data over a network and detect the state of the WLAN and the IP address of the device.

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

4.2.2.2 Layout

Since the phone application does not have a main activity, the layout of the notification that is displayed as long as the `ListenerService` is running, is discussed first. The notification consists of a `RelativeLayout` with the logo of the application on the left edge, two `TextViews` in the middle and a settings button and a close button on the right edge (Fig. 4.6a).

The top `TextView` informs the user if the phone is connected to the smartwatch. The bottom `TextView` displays a string that gives information on the current network status of the phone and thus the smartwatch. Since the the phone can connect to the UE4 plugin either via a WLAN or via hotspot, there are three possible outputs:

1. "Phone IP Address (WIFI): {The IP address of the device}"
2. "Phone IP Address (Hotspot): {The IP address of the device}"
3. "No valid connection. Please enable wifi or use a hotspot."

The close button can be used to manually stop the `ListenerService` and close the notification.

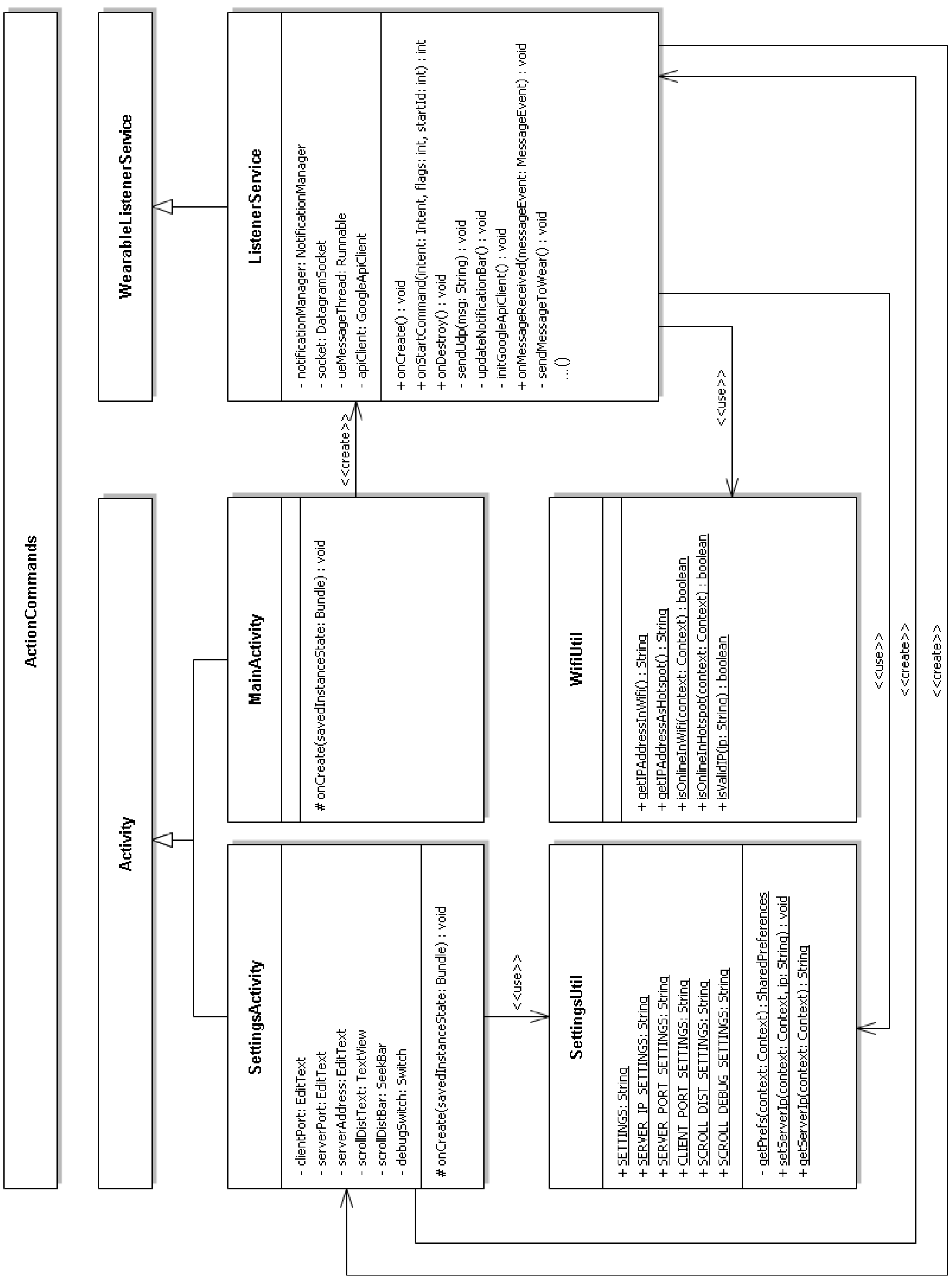
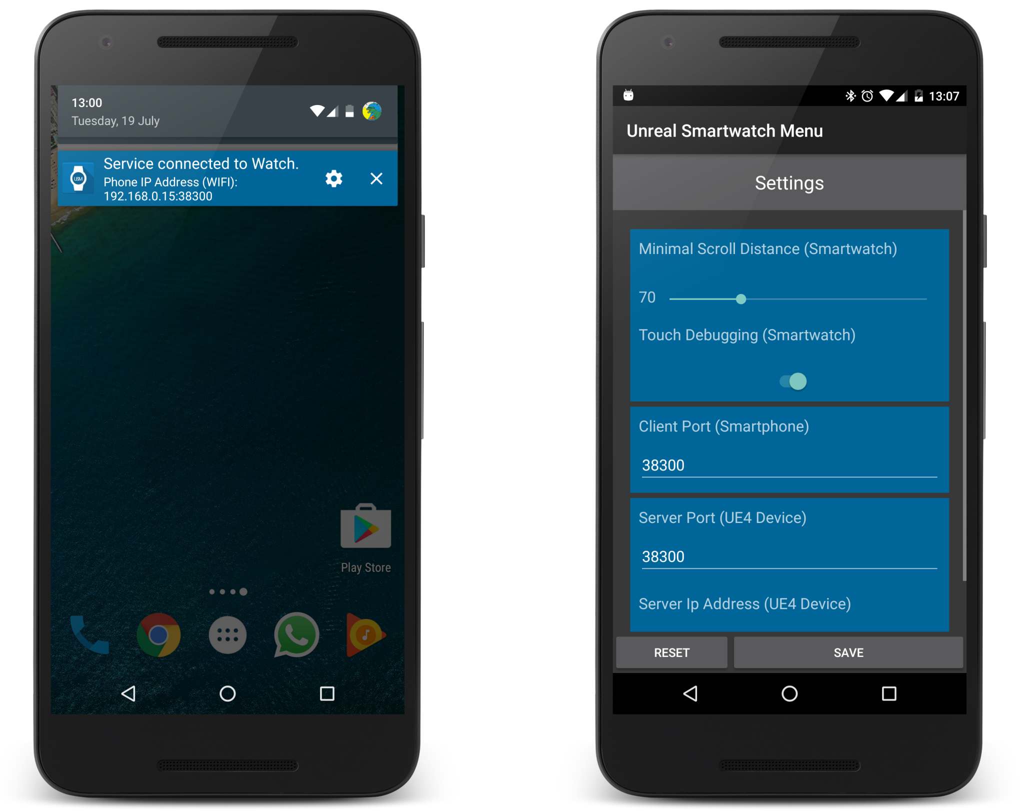

ActionCommands
WearableListenerService
Activity
ListenerService
- notificationManager: NotificationManager
- socket: DatagramSocket
- ueMessageThread: Runnable
- apiClient: GoogleApiClient
+ onCreate() : void
+ onStartCommand(intent: Intent, flags: int, startId: int) : int
+ onDestroy() : void
- sendUdp(msg: String) : void
- updateNotificationBar() : void
- initGoogleApiClient() : void
+ onMessageReceived(messageEvent: MessageEvent) : void
- sendMessageToWear() : void
...()
<<create>>
MainActivity
# onCreate(savedInstanceState: Bundle) : void
<<use>>
WifiUtil
+ getIPAddressInWifi() : String
+ getIPAddressAsHotspot() : String
+ isOnlineInWifi(context: Context) : boolean
+ isOnlineInHotspot(context: Context) : boolean
+ isValidIP(ip: String) : boolean
SettingsActivity
- clientPort: EditText
- serverPort: EditText
- serverAddress: EditText
- scrollDistText: TextView
- scrollDistBar: SeekBar
- debugSwitch: Switch
# onCreate(savedInstanceState: Bundle) : void
<<use>>
SettingsUtil
+ SETTINGS: String
+ SERVER_IP_SETTINGS: String
+ SERVER_PORT_SETTINGS: String
+ CLIENT_PORT_SETTINGS: String
+ SCROLL_DIST_SETTINGS: String
+ SCROLL_DEBUG_SETTINGS: String
- getPrefs(context: Context) : SharedPreferences
+ setServerIp(context: Context, ip: String) : void
+ getServerIp(context: Context) : String
<<use>>
<<create>>
<<create>>

Figure 4.5: UML Class Diagram of the Phone Application with the most important class variables and methods.


13:00
Tuesday, 19 July
Service connected to Watch.
Phone IP Address (WIFI):
192.168.0.15:38300

13:07
Unreal Smartwatch Menu
Settings
Minimal Scroll Distance (Smartwatch)
70
Touch Debugging (Smartwatch)
Client Port (Smartphone)
38300
Server Port (UE4 Device)
38300
Server Ip Address (UE4 Device)

(a) Notification (b) Settings Activity

Figure 4.6: Smartphone Application.

When the settings button is pressed from the notification, the `SettingsActivity` is opened up (Fig. 4.6b) and the `ListenerService` including the notification is closed. The layout for the settings contains a `ScrollView` filed with different settings cards and a reset and a save button on the bottom.

The settings cards let users choose the *minScrollDistance* via a `SeekBar`, whether or not visual debugging is enabled on the watch with the help of a `Switch` and the different ports and the address via `TextViews`. The functionality of the buttons is discussed in the next section.

4.2.2.3 Functionality

`ListenerService` and `WifiUtil`

In the *OnCreate()* method, the *GoogleApiClient* is built first for communication with the watch. As soon as the smartwatch node ID is appointed, the phone sends the *Action Commands* for minimum scroll distance and visual debugging to the watch.

Then the notification is prepared and shown with the help of the `NotificationManager`. The notification gets updated every few seconds, to give information on the current network connection. To get this information, the `WifiUtil` class is used. It provides static methods to check if the phone is connected to the internet via a WLAN or a hotspot, get the IP address in that network and finally check if an IP is valid.

After that, the UDP socket is created in the `ListenerService` and bound to the port specified in the settings. As soon as the socket is created, the *ueMessageThread* starts running that listens for `DatagramPackets` coming in on the socket. A `Thread` is a sequential process of executions. When an application is launched, the system creates a main thread for this application. The VM of Java however allows applications to have multiple concurrent threads of execution that can be defined by the developer.

When a message is received from the connected wearable device and it is a valid *Action Command* that should be sent to UE4 plugin, it gets forwarded via the socket. To send messages via UDP, the string message gets converted into a byte array and is then sent as a `DatagramPacket` to the IP address and port specified in the settings.

When the application is closed, either through an *Action Command* from the watch, from the UE4 plugin or via the close button on the notification, the *ueMessageThread* is stopped and the socket and the notification are closed.

`SettingsActivity` and `SettingsUtil`
As soon as the `SettingsActivity` is opened, the different settings fields get filled with the saved data with the help of the `SettingsUtil`. In this utility class, `SharedPreferences` are used to store and retrieve preference data as key-value sets. This means that a specific key, for example *SCROLL_DEBUG_SETTINGS* is saved with a corresponding value, which in this case would be a boolean value — true or false.

The two buttons on the bottom of the `SettingsActivity` each have a custom `OnClickListener` attached that is called as soon as the button is pressed. When the save button is pressed, the values in the different setting fields are validated. The integer of a port needs to be between 0 and 65535 and the string of the IP address is validated with a method in the `WifiUtil`. After the validation and storing of the values, the `SettingsActivity` is closed and the `ListenerService` is started again. When the reset button is pressed, all settings are set to predefined values. The standard ports are 38300 since those are most likely not occupied by other programs. The standard IP address is 192.168.0.3, since 192.168.0.1 is the default IP address used by some routers. The standard minimum scroll distance on the watch is 50 since it felt the most natural during testing and the visual debugging option is enabled.

4.3 Unreal Engine 4 Plugin

In this section the implementation of the UE4 plugin is presented. During the creation of the plugin, different versions of the UE4 source (4.10 to 4.12.5) were used since the update rate by Epic Games is relatively high. The naming of the application created with the UE4 editor is of no importance, since only independent code is packaged as a plugin.

The plugin consists of multiple classes (Fig. 4.7) which are discussed separately. First the implementation of the `UserWidget` derived class `WatchMenu` is described. Afterwards the `UDPSocketService` and the different menu components are discussed.

At the end of the section there is a short explanation on the packaging of the plugin.

4.3.1 WatchMenu

The `UWatchMenu` is the class that is used as a parent for the VR menu blueprint in the UE4 editor and handles the navigation across the different components. The class posses four different public *UPROPERTY* variables: *ServerPort*, *ClientIpAddress*, *ClientPort* and *DisplayDebugMessages*. This property adds in-editor modification support with the *EditAnywhere* specifier.

The `UWatchMenu` ticks every frame to update itself. On the first tick, the menu is initialized and the four variables specified in the editor are validated with the same constraints as discussed in section 4.2.2.3. After that, the `UWidgetTree` of the `UUserWidget` is iterated through with the help of the *UWidgetTree::ForEachWidget()* method. This tree contains all widget components that are added in the editor. Each widget that has a `IWatchMenuComponent` interface is initialized and then added to the *AllMenuContent* array.

After that, the `UDPSocketService` is initialized and a receive timer is started with the `TimerManager`. This timer is set to call the *ReceiveMessage* method of the socket service every few milliseconds. When a non-empty message is received, it is checked if there is more than one *Action Command* in the message. This can happen due to fast scrolling on the watch. If there is more than one command, the message is split up in separated commands that are handled consecutively.

On an activate command, the `ESlateVisibility` of the menu is set to visible and the *UFUNCTION OnOpened()* is called. This method does not have an implementation in code since it is declared as *BlueprintImplementableEvent* which means that it can be implemented in the visual scripting graph in the editor to further customize the event of opening the menu. When a deactivate command is received, the menu is hidden and the *OnClosed()* method is called.

On a click command, the *ClickAction()* method defined in the `IWatchMenuComponent` interface of the currently selected component is called. Additionally `UWatchSliders`, `UWatchComboBoxes` and `UWatchSpinBoxes` are focused so that further scroll commands change the values of the components instead; such as for example the slider of the `UWatchSliders`. After another click command is received, the component looses focus and normal navigation is possible again.

On a scroll command, the *Navigate(EUINavigation Direction)* method is called with the corresponding direction. The method first checks if the widget that is navigated to is a valid `IWatchMenuComponent` or if there even is any custom navigation provided for the current component. If there is not, then the regular navigation technique is used

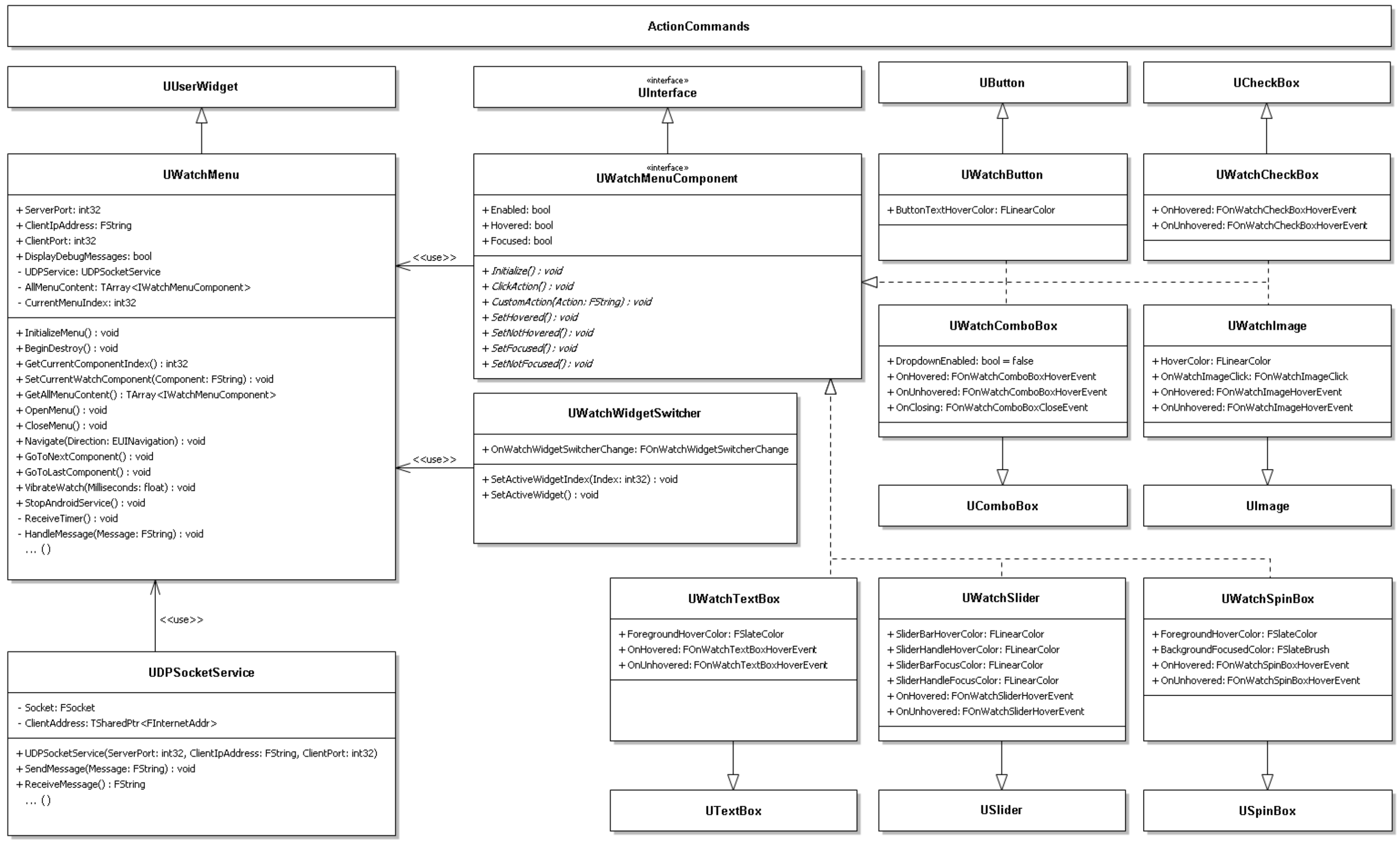

ActionCommands
UUserWidget
«interface»
UInterface
UButton
UCheckBox
UWatchMenu
+ ServerPort: int32
+ ClientIpAddress: FString
+ ClientPort: int32
+ DisplayDebugMessages: bool
- UDPService: UDPSocketService
- AllMenuContent: TArray<IWatchMenuComponent>
- CurrentMenuIndex: int32
+ InitializeMenu() : void
+ BeginDestroy() : void
+ GetCurrentComponentIndex() : int32
+ SetCurrentWatchComponent(Component: FString) : void
+ GetAllMenuContent() : TArray<IWatchMenuComponent>
+ OpenMenu() : void
+ CloseMenu() : void
+ Navigate(Direction: EUINavigation) : void
+ GoToNextComponent() : void
+ GoToLastComponent() : void
+ VibrateWatch(Milliseconds: float) : void
+ StopAndroidService() : void
- ReceiveTimer() : void
- HandleMessage(Message: FString) : void
... ()
<<use>>
«interface»
UWatchMenuComponent
+ Enabled: bool
+ Hovered: bool
+ Focused: bool
+ Initialize() : void
+ ClickAction() : void
+ CustomAction(Action: FString) : void
+ SetHovered() : void
+ SetNotHovered() : void
+ SetFocused() : void
+ SetNotFocused() : void
UWatchButton
+ ButtonTextHoverColor: FLinearColor
UWatchCheckBox
+ OnHovered: FOnWatchCheckBoxHoverEvent
+ OnUnhovered: FOnWatchCheckBoxHoverEvent
UWatchComboBox
+ DropdownEnabled: bool = false
+ OnHovered: FOnWatchComboBoxHoverEvent
+ OnUnhovered: FOnWatchComboBoxHoverEvent
+ OnClosing: FOnWatchComboBoxCloseEvent
UWatchImage
+ HoverColor: FLinearColor
+ OnWatchImageClick: FOnWatchImageClick
+ OnHovered: FOnWatchImageHoverEvent
+ OnUnhovered: FOnWatchImageHoverEvent
UWatchWidgetSwitcher
+ OnWatchWidgetSwitcherChange: FOnWatchWidgetSwitcherChange
+ SetActiveWidgetIndex(Index: int32) : void
+ SetActiveWidget() : void
<<use>>
UComboBox
UImage
<<use>>
UDPSocketService
- Socket: FSocket
- ClientAddress: TSharedPtr<FInternetAddr>
+ UDPSocketService(ServerPort: int32, ClientIpAddress: FString, ClientPort: int32)
+ SendMessage(Message: FString) : void
+ ReceiveMessage() : FString
... ()
UWatchTextBox
+ ForegroundHoverColor: FSlateColor
+ OnHovered: FOnWatchTextBoxHoverEvent
+ OnUnhovered: FOnWatchTextBoxHoverEvent
UWatchSlider
+ SliderBarHoverColor: FLinearColor
+ SliderHandleHoverColor: FLinearColor
+ SliderBarFocusColor: FLinearColor
+ SliderHandleFocusColor: FLinearColor
+ OnHovered: FOnWatchSliderHoverEvent
+ OnUnhovered: FOnWatchSliderHoverEvent
UWatchSpinBox
+ ForegroundHoverColor: FSlateColor
+ BackgroundFocusedColor: FSlateBrush
+ OnHovered: FOnWatchSpinBoxHoverEvent
+ OnUnhovered: FOnWatchSpinBoxHoverEvent
UTextBox
USlider
USpinBox

Figure 4.7: UML Class Diagram of the UE4 Plugin with the most important class variables and methods.

where the next viable menu component is chosen when a *right* or *down* scroll, and the last when a *left* or *up* scroll happens. *Next* components are defined as components that are vertically down in the `UWidgetTree` and *last* components are defined as vertically up in the `UWidgetTree`.

When a `UWatchTextBox` is clicked, the request voice *Action Command* is sent to the smartwatch which replies with a voice result. As long as no result or error message is received, the text box keeps its focus.

Other *UFUNCTION* methods declared as *BlueprintCallable* are the *StopAndroid-Service()* method which sends an *Action Command* that closes the smartphone and smartwatch application and the *VibrateWatch(float Milliseconds)* method that sends the command to vibrate the watch for a given amount of time. There are also methods to navigate the menu manually: *GoToNextComponent(bool Wrap)*, *GoToLastCompo-nent(bool Wrap)* and *Navigate(EUINavigation Direction)*. With the *SetCurrentWatch-Component(FString Component)* method, the developer can set the currently hovered component by providing the name of the component. All these methods can be called from the editor in the visual scripting graph.

When the UE4 application is closed, the *BeginDestroy()* method is called which destroys the `UWatchMenu` and deletes the `UDPSocketService`.

4.3.2 UDPSocketService

The `UDPSocketService` provides methods for sending and receiving messages from the Android phone.

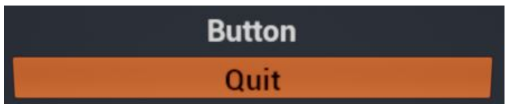
As soon as the constructor is called, the *ClientAddress* is initialized as *FInterne-tAddr* with the provided IP and port. Then the socket is created with the help of the `FUdpSocketBuilder` and bound to the port specified in the `WatchMenu`.

The *SendMessage(FString message)* method converts an `FString` to a 8-bit unsigned integer array and sends it to the *ClientAddress*. The *ReceiveMessage* method listens on the *ClientAddress* socket for any pending chunk of data and returns it after converting it to a `FString`.

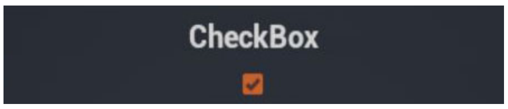
When the `UDPSocketService` object is destroyed, the socket is closed and specifically destroyed with *ISocketSubsystem::DestroySocket()* to make it available for further usage.

4.3.3 Menu Components

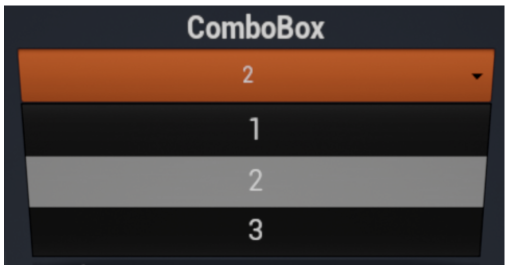
In this section the different menu components are discussed. First the the interactive components are described, afterwards components that the user can not interact with directly but have to be considered, to make menu navigation with a network commands possible. Lastly the behaviour of popups on widget components in world space in the current UE4 version is discussed.



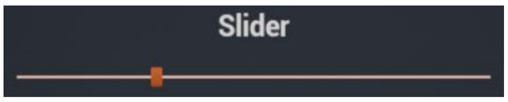
(a)



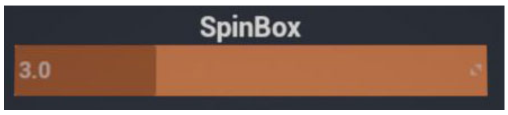
(b)



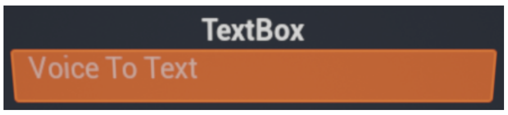
(c)



(d)



(e)



(f)

Figure 4.8: Custom Menu Components.

4.3.3.1 IWatchMenuComponents

Every interactive component in the smartwatch controllable menu has to implement the `IWatchMenuComponent` interface and thus override the methods provided by it.

When a component is hovered in the `UWatchMenu`, the *SetHovered()* method is called which, depending on the component, changes different colouring options to highlight it. For example in the hover method in the `UWatchButton` in Fig. 4.8a, the `WidgetStyle` of the underlying `UButton` is changed to an orange *ButtonHoverColor*. When the button is un-hovered, the color changes back to *ButtonNormalColor*. Additionally the color of the text on the button is changed to *ButtonTextHoverColor* or to *ButtonTextNormalColor*. Some color or style variables are already implemented in the base classes of the different menu components, there are however some missing that are missing. The text of the normal button widget for example, does not support different colors depending on a hovering event. This is why *ButtonHoverColor* is implemented as *UPROPERTY*. This enables the developer to customize components further.

Furthermore when a component is hovered or unhovered, the corresponding *OnHovered* or *OnUnhovered* events are broadcasted. Such events are declared as dynamic multicast delegates. Delegates in general make it possible to call member functions on objects in a generic but type-safe way. One benefit of delegates is that they can be triggered remotely either in the editor with a visual scripting node or in code. As with the color variables, some components already possess the necessary delegates in the base class for the developer to be able to react to different events; and some need to be added in the derived class. An example is the `UWatchComboBox` that broadcasts an *OnOpening* event but not an *OnClosing* event.

When the *ClickAction()* method is called, each component triggers different events and the custom click implementation is executed. The `UWatchButton` for example broadcasts the *OnClicked* event whereas the `UWatchCheckBox` (Fig. 4.8b) broadcasts the *OnCheckStateChanged* after toggling its checked state. There are also components that get focused as soon as they are clicked. The *SetFocused()* and *SetNotFocused()* methods are only implemented for such focusable components. Focusable components have additional color variables that can be changed by the developer to differentiate focus and highlight. When such components are clicked, further input commands trigger custom actions which are implemented in the *CustomAction(FString Action)* method:

- `UWatchComboBox` (Fig. 4.8c)
  When focused, scroll commands select the next or the last option by incrementing or decrementing the current option index. If the *DropdownEnabled* variable is set to *true*, the drop-down popup opens and scroll up and scroll down commands are used to navigate the options. If it is set to *false*, the drop-down does not pop up and navigation between items is performed with scroll left and scroll right commands. The reason for the possibility of disabling the drop-down popup is discussed in section 4.3.3.3.
- `UWatchSlider` (Fig. 4.8d)
  When focused, scroll commands either add the *StepSize* to or subtract the *StepSize* from the value of the slider. Depending on the orientation of the slider (*EOrientation::Orient_Horizontal* or *EOrientation::Orient_Vertical*), either scroll left and right or scroll up and down commands change the value.
- `UWatchSpinBox` (Fig. 4.8e)
  When focused, scroll left and scroll right commands change the value of the spin box. The *Delta* of the spin box is the pendant to the *StepSize* of the slider.
- `UWatchTextBox` (Fig. 4.8f)
  When focused, the user can speak to enter a text via the smartwatch. When the voice result command is received, the string attachment is set as text of the `UWatchTextBox` and the *OnTextCommitted* event is broadcasted. This is the only component that automatically looses focus upon receiving a command from the watch.

When a focusable component is clicked again, it becomes unfocused and normal navigation through menu components is possible again.

4.3.3.2 Panel Components

During the initialization of the `UWatchMenu`, it is determined whether or not the menu contains any `UScrollBoxes`, and additionally any `UWatchWidgetSwitchers` that the `UWatchMenu` contains are added to an array.

`UScrollBoxes` are panel components that are scrollable. When the menu contains such a component, it has to be considered that after a navigational command, a component could be outside of the visible scrolling area. Thus after the navigational methods *Navigate(EUINavigation Direction)*, *GoToNextComponent(bool Wrap)* or *GoToLast-*

Component(bool Wrap) are called, the currently selected component gets scrolled into view automatically.

`UWatchWidgetSwitchers` are panel components that make it possible to have multiple overlapping panels in an `UWatchMenu`, like a tab control without providing tabs. The switching of tabs is performed with the *SetActiveWidgetIndex(int32 Index)* method or the *SetActiveWidget(UWidget* Widget)* method. After those methods are called, the custom *OnWatchWidgetSwitcherChange* event is broadcasted from the `UWatchWidgetSwitchers` and triggers the *UpdateWidgetSwitcherIndex* method in the `UWatchMenu`. This is achieved by adding the *UpdateWidgetSwitcherIndex* method to the event during initialization with *AddDynamic(UserObject, FuncName)*. When the displayed widget in the `UWatchWidgetSwitchers` is changed, the navigable menu components have to be updated to only those that are visible.

Other panel components such as the `UCanvasPanel`, `UHorizontalBox` or `UOverlay` do not have a special impact on the `UWatchMenu` and therefore do not have to be further customized or considered.

4.3.3.3 Popups on Widget Components

During the process of implementation, UE4 version 4.10 to 4.12 were used and supported. In versions prior to 4.12, widget components that produce popups were not compatible with `UUserWidgets` in world space. Popups include the drop-down menus of `UComboBoxes`, widgets that are placed with the `UMenuAnchor` over other widges and tool tips in general. With version 4.12, *Epic Games* started to address the issues and started to work on a fix.

During multiple hot-fixes in versions 4.12.1 to 4.12.5, the proper functionality of popups was not given. Due to this unstable behaviour, the *DropdownEnabled* option is provided in the `UWatchComboBox` and `UMenuAnchors` do not support popups with smartwatch interactable content.

4.3.4 Plugin Generation

To package the code into a plugin, a plugin folder has to be created with the following structure:

```
PluginName
    Resources
        Icon128.png
    Source
        PluginName
            Private/Public
                ...
            PluginName.Build.cs
            PluginName.cpp
    PluginName.uplugin
```

The name of the plugin is *UnrealSmartwatchMenu*. As *Icon128.png*, the icon of the Android application displayed in Figure 4.2 is used. The source code has to be placed into the corresponding *Private* and *Public* folders. The *.Build.cs* and the *.cpp* files are generated by the UE4 environment and have to be added to build the plugin.

The most important content of the *.uplugin* descriptor file is presented in listing 4.2. *FileVersion* describes the current version of the descriptor file which is used for backwards compatibility. *FriendlyName* is the name of the plugin which is, same as the *Description* and *CreatedBy* fields, displayed in the plugin UI in the editor. *IsBetaVersion* tags the plugin as a beta plugin. Since world space menus are still marked as experimental, the plugin is marked as beta plugin. *Modules* contains a list of modules that should be loaded at startup. In this case, the *UnrealSmartwatchMenu* module is declared as runtime module, meaning that it should be loaded during runtime and when the editor is running.

```
{
        "FileVersion": 3,
        "FriendlyName": "Unreal Smartwatch Menu",
        "Description": "Adds custom UMG Widget Components that can be
           controlled with a smartwatch.",
        "CreatedBy": " Schuster Florian",
        "IsBetaVersion": true,
        "Modules": [
                {
                        "Name": "UnrealSmartwatchMenu",
                        "Type": "Runtime",
                }
        ]
        ...
}
```

Listing 4.2: Plugin Module Asset File.

Afterwards, the plugin is built with the UnrealBuildTool (UBT) via Visual Studio and the plugin is ready to be imported into another project. For that, the compiled folder has to be placed in the *Plugins* folder of a UE4 project.

4.4 Usage

In this section, the usage of the system is explained. First basic instructions for VR menu developers and then for Android smartwatch users are given.

4.4.1 Plugin Developer

After the developer creates a new UE4 project and adds the *UnrealSmartwatchPlugin* to the plugin folder, a new blueprint has to be created that possesses the `UWatchMenu` as parent. In the blueprint, the developer has to first create the layout of the menu.

To search for a specific widget component, the developer has to use the *Palette* panel (Fig. 4.9, 1). All components that can be either navigated by the smartwatch or are related to the plugin have a preceding `Watch-` in their name. The desired component is then dragged either to the *Hierarchy* (Fig. 4.9, 2) where the `UWidgetTree` is displayed or directly to the *Designer View* (Fig. 4.9, 3).

Additional details for each component can be customized in the *Details* panel (Fig. 4.9, 4). Here the appearance as well as the relative size can be adjusted. At the bottom of the

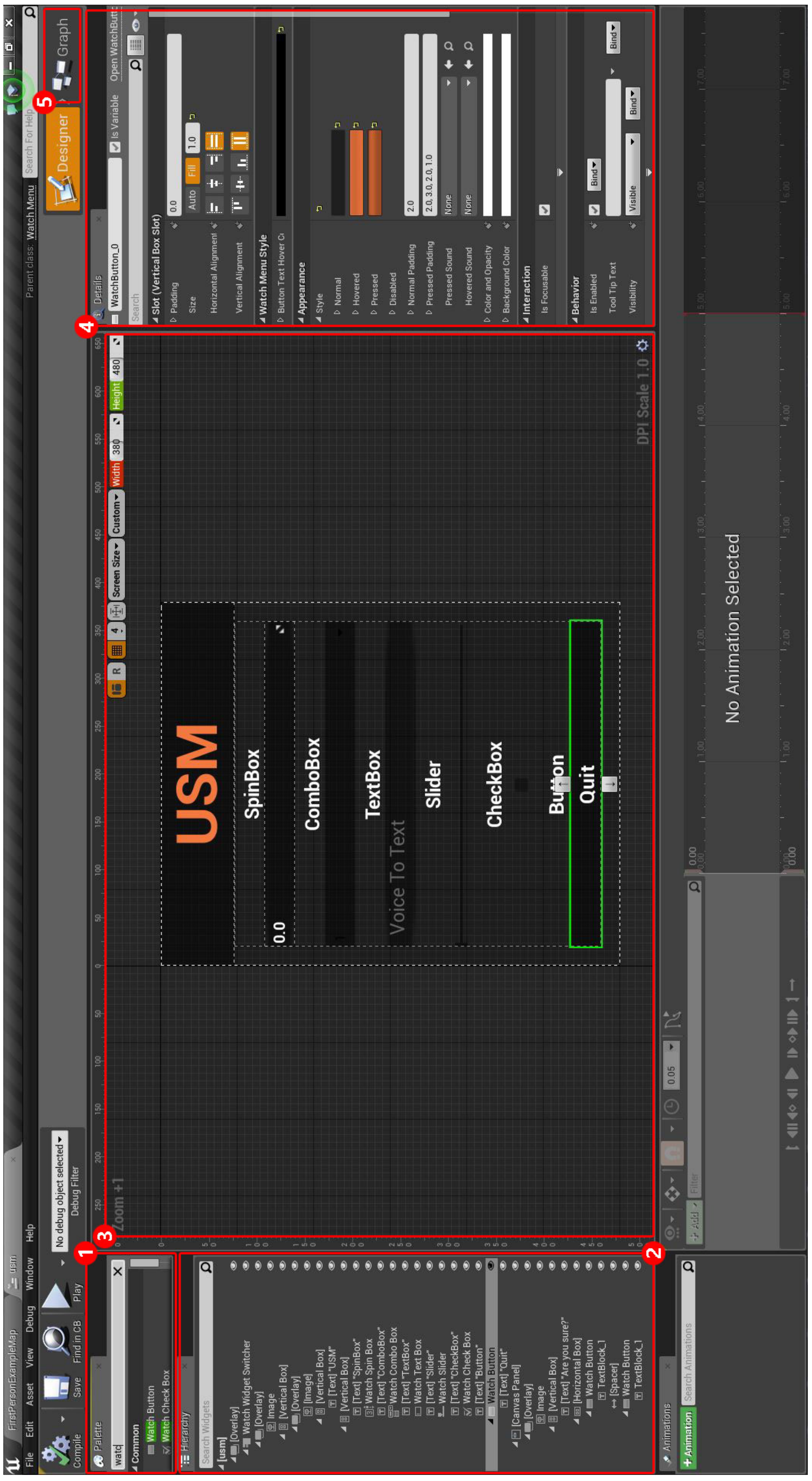


Figure 4.9: UWatchMenu Blueprint Window.

panel there is the event section where each event can be clicked to be further customized in the *Graph View* (Fig. 4.9, 5). In Fig. 4.10 the *OnClicked* event of a `UWatchButton` is customized in the *Graph View* to vibrate the watch for 200 milliseconds and then switch the index of the *WatchWidgetSwitcher0* to 1, meaning that the the active widget as seen in Fig. 4.11a is changed to the widget seen in Fig. 4.11b.

In the *Details* panel, there is also the option to customize navigation of each component (Fig. 4.12). The standard navigation rule is declared as *EUINavigationRule::Escape* where on left and up scrolls the previous and on right and down scrolls the next menu component is selected. On *EUINavigationRule::Stop*, the navigation in the given direction is stopped. With the *EUINavigationRule::Wrap* rule, a scroll down command on a component that is at the bottom of the menu, results in the topmost component getting hovered and vice versa. This wrapping inside the container only works on vertical menus however, which makes the *EUINavigationRule::Explicit* navigation rule a must for menus that enable the user to continuously scroll over bounds. With this rule, the developer can set a fixed component to move to for a certain direction of navigation. Since the watch sends left, right, up and down commands, the next and previous directions are not supported.

Finally, the developer can edit the IP and port settings by clicking the topmost widget in the *Hierarchy* which is the menu itself. The connection settings can then be customized in the *Details* panel.

After the menu is created in the blueprint, it has to be added to the game. This can be achieved by either placing it in the blueprint of the player controlled actor, in a separate actor that is static, etc.

4.4.2 Android User

Firstly, users have to make sure that their phones are connected to the WLAN that the device of the VR menu is connected to. If that is not possible, then a hotspot has to be created and the UE4 device has to be connected with it.

After installing the APK of the application on the smartphone and thus automatically on the paired smartwatch, the user has to start the *USM* application on the watch. After that the connection settings have to be edited to align them with the settings of the VR menu by pressing the settings button to open the settings window.

After this one time only connection setup, the user can close or open the application at any time when connected with the same WLAN as the UE4 device to control the VR menu.

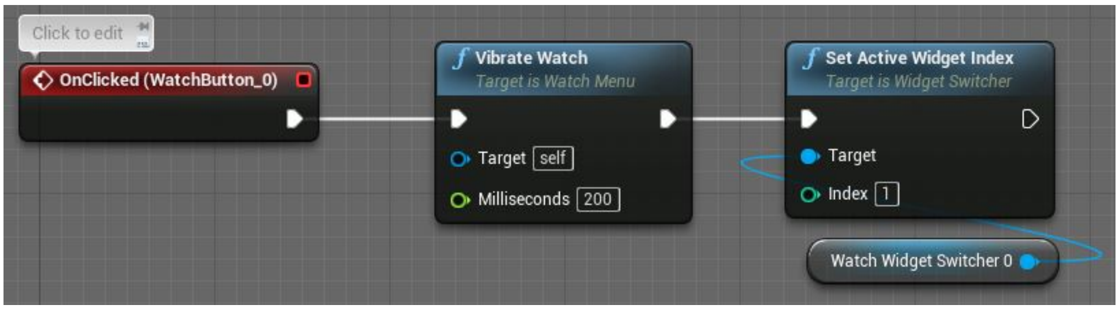
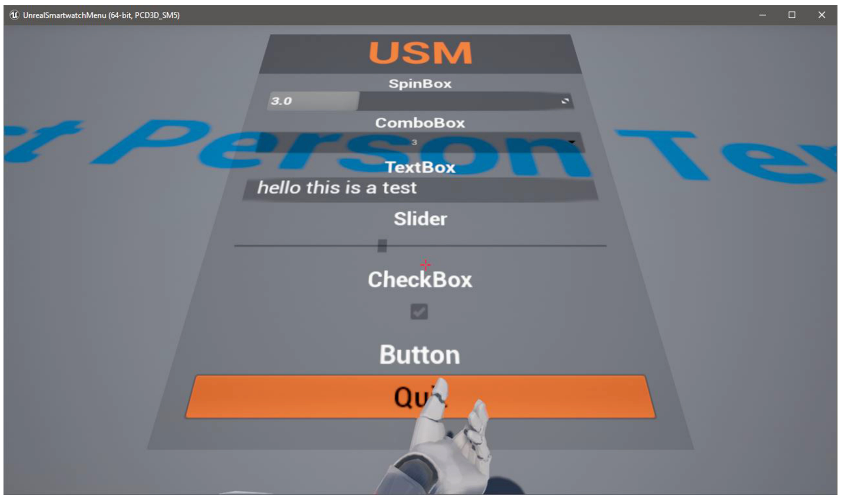
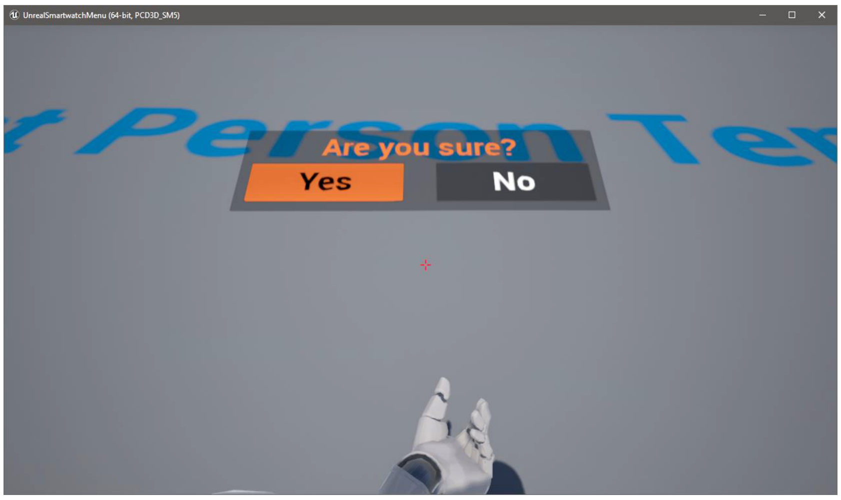


Figure 4.10: Event Customization in the *Graph View*.

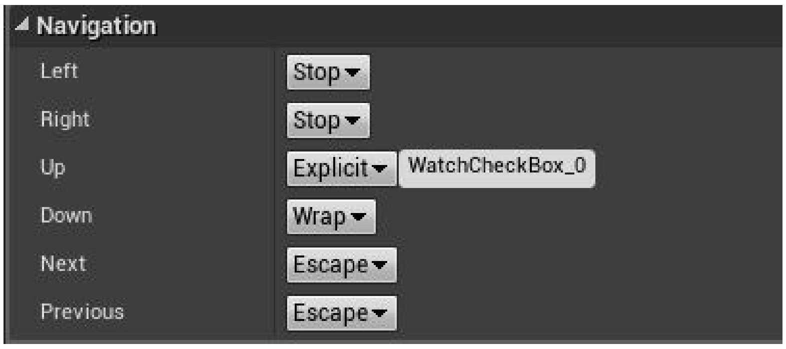


(a)



(b)

Figure 4.11: A sample `UWatchMenu` in a Game.



Navigation

Left	Stop
Right	Stop
Up	Explicit WatchCheckBox_0
Down	Wrap
Next	Escape
Previous	Escape

Figure 4.12: Navigation Customization in the *Details Panel*.

CHAPTER 5

Results

In this chapter, the results of the implemented system are presented. First, four examples of VR menus created with the plugin are presented. Afterwards, the fulfilment of the performance requirements are reviewed for each device separately.

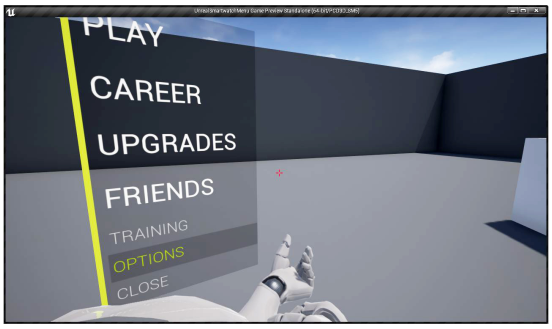
5.1 Virtual Reality Menus Examples

5.1.1 USM Menu

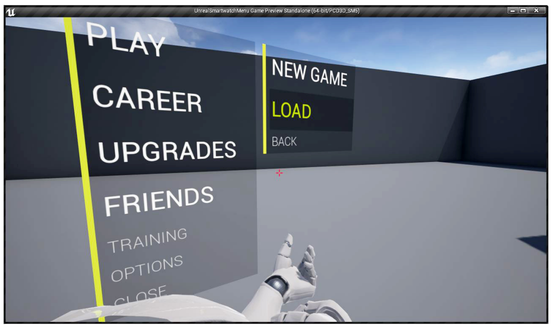
The *first* menu created with the help of the UE4 plugin is the *USM Menu* (Fig. 4.11 and Fig. 4.9). It is a simple vertical menu that is displayed in front of the user on the ground and moves and turns with the player character. It contains all available `IWatchMenuComponents`: one `UWatchSpinBox`, one `UWatchComboBox`, one `UWatchTextBox`, one `UWatchSlider`, one `UWatchCheckBox` and one `UWatchButton`. It also contains one `UWatchWidgetSwitcher` that is used to switch between *Widget0* (Fig. 4.11a) and *Widget1* (Fig. 4.11b). The widgets are switched as soon as the button is pressed (Fig. 4.10). In *Widget1* there is another smaller menu that is used as a confirmation dialogue to quit the game.

5.1.2 Game Menu

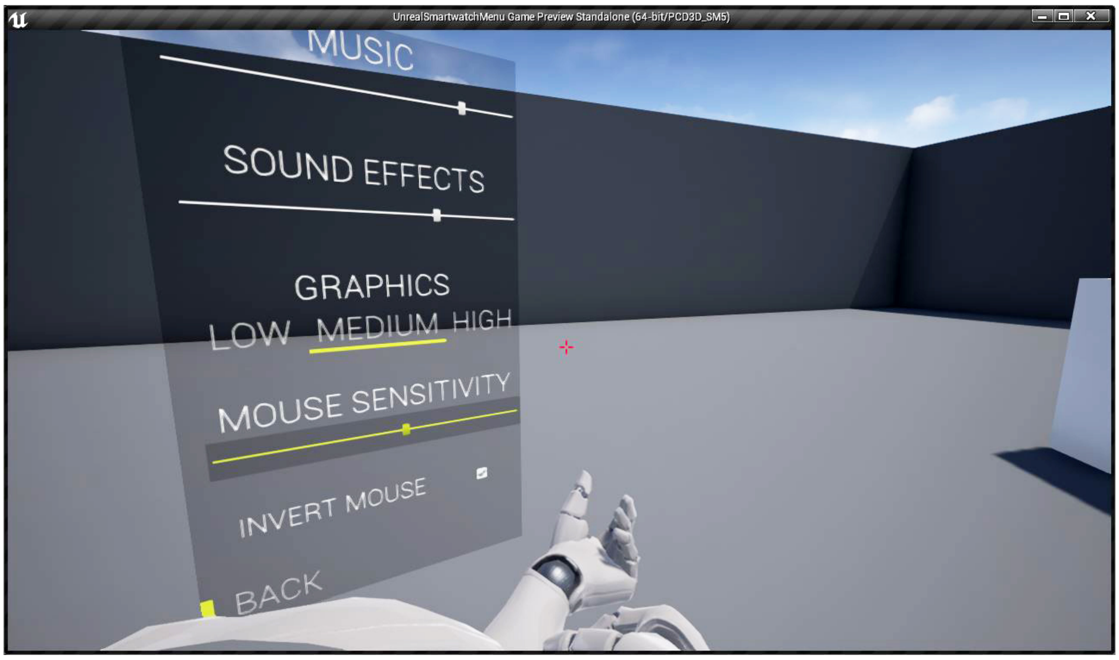
The second menu is the *Game Menu* (Fig. 5.1) that consists of two `UWatchWidgetSwitchers`. The first one switches between the normal game menu (Fig. 5.1a) and the options menu (Fig. 5.1c) while the other one switches between a blank widget and the additional widget on the side when pressing the *Play* button (Fig. 5.1b). Generally, as soon as a component is hovered, it is highlighted with a bright green color and a dark background. The menu is displayed in front of the user with a slight tilt forwards and moves and turns with the player character. The options menu contains three `UWatchSlider`, one `UWatchCheckBox` and a `UWatchButton` on the bottom to get back to the main


UnrealSmartwatchMenu Game Preview Standalone (64-bit/PCD3D_SM5)
PLAY
CAREER
UPGRADES
FRIENDS
TRAINING
OPTIONS
CLOSE

(a)


UnrealSmartwatchMenu Game Preview Standalone (64-bit/PCD3D_SM5)
PLAY
CAREER
UPGRADES
FRIENDS
TRAINING
OPTIONS
CLOSE
NEW GAME
LOAD
BACK

(b)



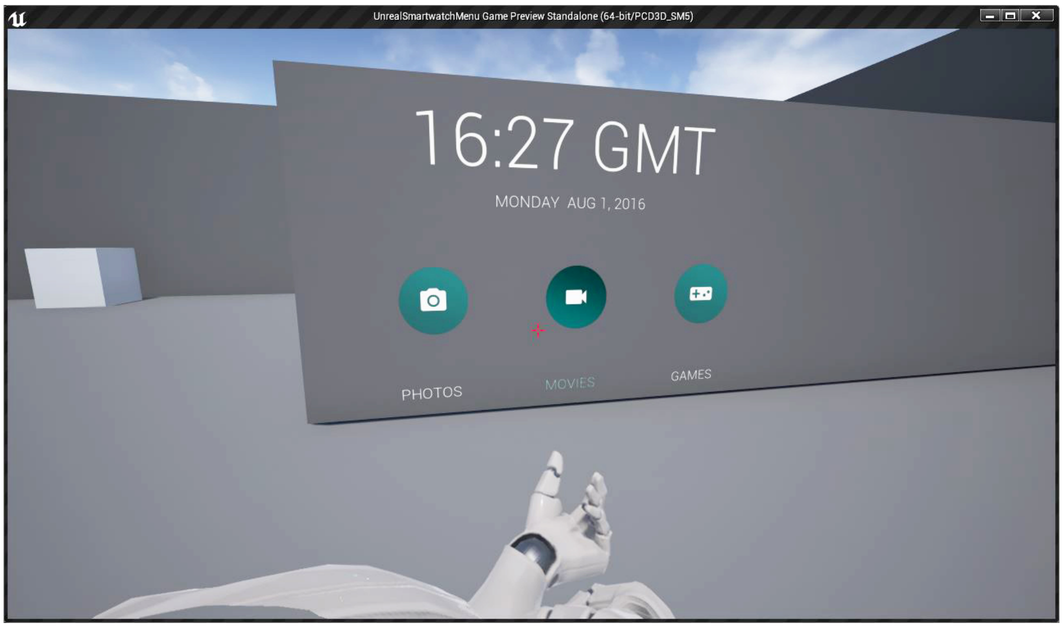
(c)

Figure 5.1: Game Menu.

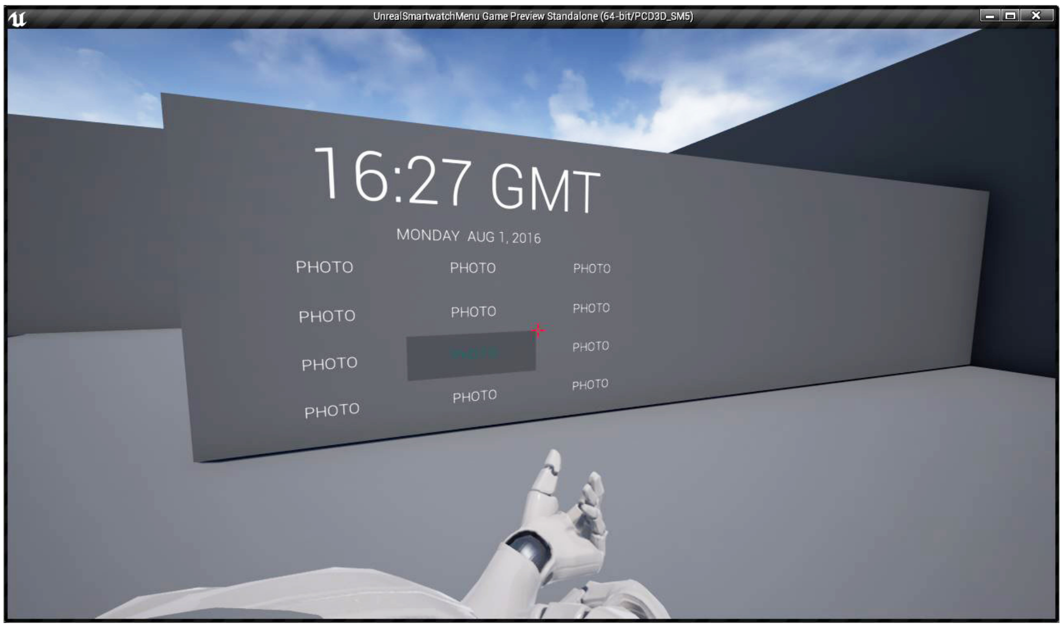
menu. The *Graphics* selection is composed of three UWatchButtons that when clicked, respectively enable or disable the visibility of the line image underneath to indicate which option is selected.

5.1.3 Home Menu

The third menu is the *Home Menu*. It should simulate a home screen menu for a VR application that shows the time, the date and general media (Fig. 5.2). The three UWatchButtons on the bottom of the first widget (Fig. 5.2a) do not contain text but instead icons. The color of the text underneath each button is changed depending on the hover state of the corresponding button. When a button is pressed, a list is opened with


UnrealSmartwatchMenu Game Preview Standalone (64-bit/PCD3D_SM5)
16:27 GMT
MONDAY AUG 1, 2016
PHOTOS
MOVIES
GAMES

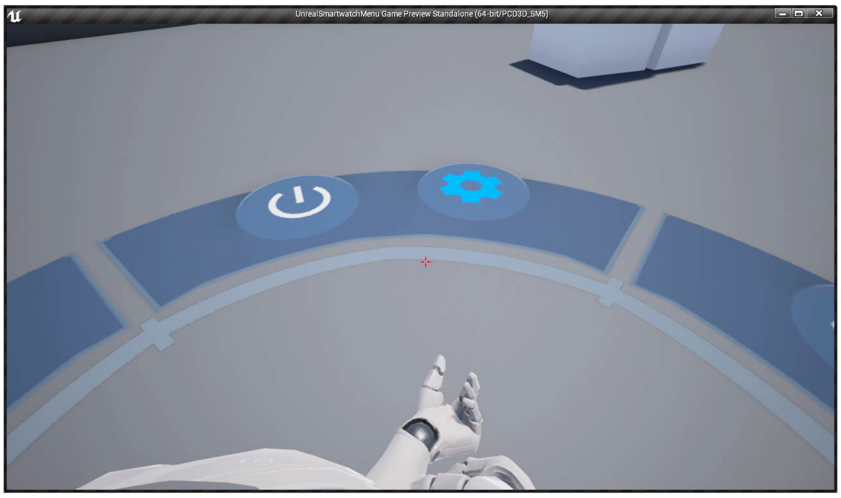
(a)



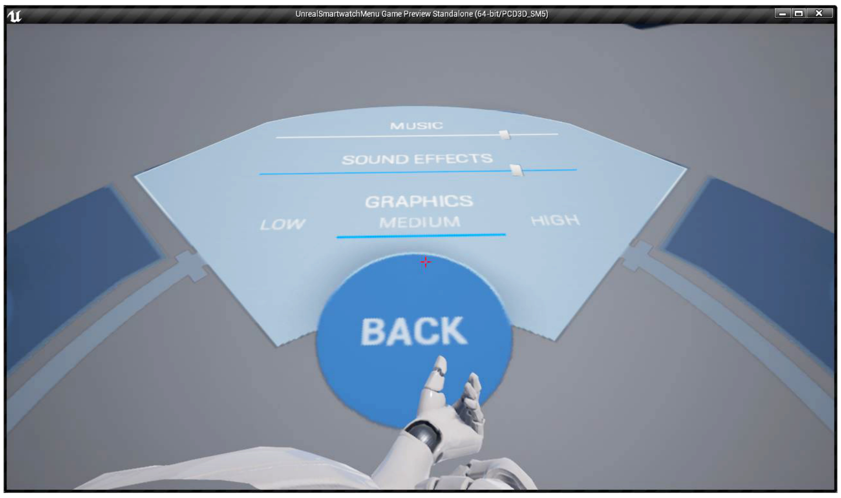
(b)

Figure 5.2: Home Menu.

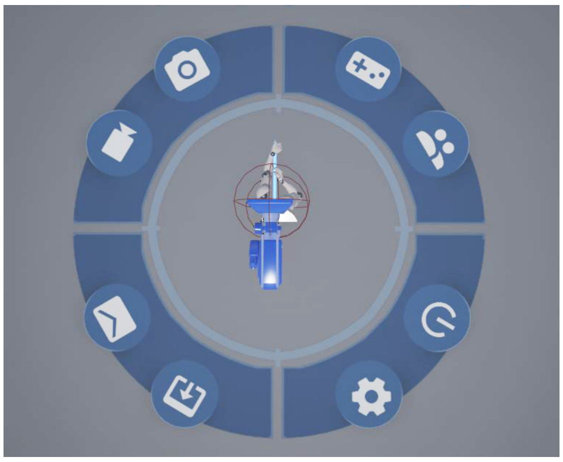
`UWatchButtons` inside a `UUniformGridPanel` that act as a mockup of real media (Fig. 5.2b).


UnrealSmartwatchMenu Game Preview Standalone (64-bit/PCD3D_SM5)

(a)


UnrealSmartwatchMenu Game Preview Standalone (64-bit/PCD3D_SM5)
MUSIC
SOUND EFFECTS
GRAPHICS
LOW
MEDIUM
HIGH
BACK

(b)



(c)

Figure 5.3: Ring Menu.

5.1.4 Ring Menu

The fourth menu is the *Ring Menu.* It consists of a ring of `UWatchButtons` that are arranged on top of a background image (Fig. 5.3c). The menu moves with the player character but does not turn. The user switches between options with left and right scrolls (Fig. 5.3a) and when selecting a choice, another widget is brought up with the help of a `UWatchWidgetSwitcher` that acts as a submenu (Fig. 5.3b).

5.2 Performance

To test the performance of the implementation and confirm compliance with the requirements, a usage test is conducted. The best way to test the implementation would be to have a test with multiple users, each having a smartwatch, that interact in an immerive VR system with a specifically developed game. However since there are no games implemented for this target setup yet, and making one is not in the scope of this thesis, a custom performance test is conducted. In two sessions of one hour each, the system is tested actively and passively.

In the *passive* test, the previously presented *USM* menu is run in a clean UE4 *First Person Template* project. It is used due to the fact that it contains all available `IWatchMenuComponents`. The Android applications are both running during the test but no input actions are performed.

In the *active* test, the *USM* menu is used as well. To simulate active use of the menu, the average frequency of menu usage in games or VR applications is needed. Unfortunately there are no statistics as of writing this thesis. As reference point, a small survey on menu usage in computer games is conducted where gamers are asked how long they approximately spend their time in game menus during a one hour gaming session. Menus include settings or options but also inventory menus or game related interfaces. The questionnaire was done online with the help of StrawPoll. The results are presented in table 5.1.

Answer	**Number of Answers**	**Percent**
< 5 minutes	25	42%
< 10 minutes	26	43%
< 20 minutes	7	12%
> 20 minutes	2	3%
Total	59	100%

Table 5.1: Menu Usage in a one hour gaming session.

Even though 85% of the respondents stated that they use menus less than 10 minutes in one hour, 20 minutes is chosen as reference value for the active test as limit case.

5.2.1 Smartwatch Resource Usage

The watch application takes up approximately 13 MB of storage space. This is determined with the help of the *Android Wear Companion Application* that displays the installed applications on the wearable device.

In table 5.2 the results for the smartwatch usage test are presented. The memory usage is monitored with the *Memory Monitor* that Android Studio provides. The difference between using the application actively or passively is 2% of the smartwatch's 512 MB RAM. Considering the fact that approximately 900 more *Action Commands* are sent in the active test, the difference is moderately low. The number of commands is measured with the help of the debugging functionality of Android Studio. The battery drainage is higher by 4% is the active test which is reasonable for a one hour session. Battery drainage could be further lessened by disabling screen debugging during the usage of the smartwatch.

	Passive	**Active**
Test Time/Active Use (Minutes)	60/0.1	60/20
***Action Commands* (W → P → UE4)**	4	903
RAM Usage (%)	62	64
Battery Drainage (%)	12	16

Table 5.2: Results of the smartwatch performance test. (W ... Watch, P ... Phone)

5.2.2 Smartphone Resource Usage

The phone application takes up approximately 23 MB of storage space, where one half is the original APK and the other half is the actually installed Dalvik Executable (DEX) file.

In table 5.3 the results for the smartphone usage test are presented. The data for memory usage and battery drainage is gathered from the *Settings → Apps → Unreal Smartwatch Menu* screen on the phone. The difference of RAM usage between tests is 0.2% which is very low. The amount of battery consumed by the application during both tests is too low to be measured, so it is assumed that it is somewhere below 1%.

	Passive	**Active**
Test Time/Active Use (Minutes)	60/0.1	60/20
***Action Commands* (P → W)**	2	102
RAM Usage (%)	6.5	6.7
Battery Drainage (%)	< 1	< 1

Table 5.3: Results of the smartphone performance test. (W ... Watch, P ... Phone)

5.2.3 Unreal Engine 4 Plugin Resource Usage

The size of the plugin folder amounts to 86 MB since it contains pre-compiled files.

In table 5.4 the results for the UE4 plugin usage test are presented. The values are gained with the help of *perfmon* which is a performance calculation tool by Windows.

The difference in RAM usage between tests is measured at 203 MB which amounts to 1.3% of the 16 GB of memory of the testing device. Additionally the CPU time of the UE4 process is measured. This value is calculated as the percentage of elapsed time that the processor spends to execute a non-idle thread and then subtracted by 100%. The percentage value in the resulting test is higher than 100% because it is the sum of time on each processor. Since the working machine has four cores, the maximum value is 400%. The difference between tests is 7.6% which amounts to 1.9% increased CPU time on all processors. Battery drainage is the same for both the passive and the active test meaning that the slightly increased CPU and RAM usage do not impact battery discharge.

	Passive	**Active**
Test Time/Active Use (Minutes)	60/0.1	60/20
***Action Commands* (UE4 → P → W)**	3	101
RAM Usage (MB)	10646	10849
CPU Time (%)	127	134.6
Battery Drainage (%)	82	82

Table 5.4: Results of the UE4 performance test. (W . . . Watch, P . . . Phone)

Both the UE4 plugin and the Android applications were not expected to be computationally expensive. In general, even though there were no fixed performance requirements, all the factors stated in the corresponding requirement sections regarding battery drainage, memory usage and storage space were considered during the implementation and kept to a minimum to prevent performance restrictions.

5.3 Network Usage

Additional to device performance, also network usage is monitored during the passive and active tests. The results are presented in table 5.5.

	Passive	**Active**
Smartphone (Byte)	128	58k
UE4 (Byte)	202	6670

Table 5.5: WLAN data sent during the performance tests.

WLAN data sent from the phone to the menu is measured with the help of the network protocol analyzer Wireshark. As capture filter *host 192.168.0.15* is used to only detect packages sent to and from the smartphone. Afterwards the following filter is used to only see UDP packages that are sent to the VR menu: *ip.dst == 192.168.0.3 and udp.length>0*. The amount of data that is sent in one hour amounts to 58 Kilobyte (KB) which is approximatley 1 KB per second. This number is comparatively low, considering that immersive VR systems such as the *ImmersiveDeck* [56] have networks that are capable of a total throughput of 1.73 GB per second.

WLAN data sent from the menu to the phone is measured with the same application, however with a different filter: *ip.dst == 192.168.0.15 and udp.length>0*. This filter only shows UDP packages that are sent to the phone. The data sent amounted to 6.7 KB an hour in the active test which is approximately 0.1 KB per second. This number, same as the one of the smartphone, is very low.

To sum up, data sent over the WLAN is the only difference between the active and the passive test. However as discussed before, this difference is still very much acceptable considering the network capabilities of immersive VR systems.

5.4 Open Issues

5.4.1 Android

At the time of the implementation of the system, the Android Wear API level was version 1.5. In version 2.0 it could be possible to have the two part Android system combined on the smartwatch, without needing a smartphone service. Due to improvements of the IMF, it could be possible to change network settings on the watch directly with the on-screen keyboard. However it would have to be considered that the smartwatch needs to be able to connect to a WLAN.

5.4.2 Unreal Engine 4

There are three main issues with 3D widgets in world space in UE4. These issues can only be corrected by Epic Games by updating the engine itself:

1. Popups are unstable in the current version (see section 4.3.3.3).
2. UMG menus in UE4 are not really *3D Widgets* since they are only *Flat Menus (Mapped on Geometry)*. To have real three dimensional widgets, Epic Games would have to disregard UMG and create another system that is not based on slate.
3. They are marked as experimental, with the annotation that they theoretically could be substantially changed in future releases of the engine.

One open issue exists in the implementation of the plugin. The biggest disadvantage by binding one smartwatch to one `UWatchMenu` is that only this menu can be controlled with the watch. It is for example not possible to simultaneously navigate a world space and a screen space menu with only one watch since each menu is encapsulated. A possible fix for this problem could be to create a custom input controller mechanism that works like the input of a joystick or other VR input devices like the Leap Motion or the controllers of the HTC Vive. This input would work globally and one could be able to focus different menus. However the native implementation of different widgets in UE4, make it very difficult to select, hover, focus or interact with standard components without the help of mouse input or without deriving from the base component classes.

CHAPTER 6

Conclusion

In this thesis, a correlating system of two components, consisting of an UE4 plugin and an Android application was implemented and presented. The plugin enables developers to create UMG menus in the graphical interface of the UE4 editor whereas the Android application enables users of an immersive VR system to interact with the menus via touch input on a smartwatch over a WLAN.

The Android application consists of a smartwatch part and a smartphone part. The smartwatch part is used to input specific interaction commands that get sent to the smartphone part over bluetooth. The smartphone part then sends the commands on a WLAN to the VR menu in UE4. The UE4 plugin allows a developer to use custom UMG widget components to create a smartwatch navigable menu. The menu receives the UDP packages from the smartphone and then translates them into menu navigation.

Test results show that neither the Android applications, nor the UE4 plugin have a considerable performance impact on the respective devices. To show the possibilities of the plugin, four VR example menus were implemented that can be navigated with a smartwatch.

In future work, the implemented system could be used to create a VR menu that is fully integrated into an immersive VR system, whereby it could be tested with multiple users.

List of Figures

List of Tables

Acronyms

ADB	Android Debug Bridge. 43, 44
AMOLED	Active-Matrix Organic Light-Emitting Diode. 26
API	Application Programming Interface. 20, 23, 24, 26, 39, 66
APK	Android Application Package. 43, 56, 64
AR	Augmented Reality. 14
ATAP	Advanced Technology and Projects. 16
CDO	Class Default Object. 33
CPU	Central Processing Unit. 24, 27, 65
CV	Consumer Version. 1
DEX	Dalvik Executable. 64
DK2	Developer Kit 2. 6, 14
DOF	Degree Of Freedom. 12, 14–16
FPS	First-Person Shooter. 28
GB	Gigabyte. 22, 24, 30, 65, 66
GPS	Global Positioning System. 5
GUI	Graphical User Interface. 11
HCI	Human Computer Interaction. 1, 10, 13
HMD	Head Mounted Display. ix, xi, 1, 2, 5–8, 23, 25
HUD	Head-up-Display. 28

ID Identifier. 42, 47

IDE Integrated Development Environment. 21, 22, 31, 43

IMF Input Method Framework. 24, 66

IMU Inertial Measurement Unit. 5, 7, 13, 14, 16, 24

IP Internet Protocol. 27, 28, 34, 44, 47, 50, 56

IPS Internet Protocol Suite. 34

IR Infrared. 15, 16

KB Kilobyte. 66

LCD Liquid-crystal Display. 14, 17, 26

mAh Milliampere-Hour. 24

MB Megabyte. 22, 64, 65

MEMS Micro-Electro-Mechanized-System. 14

MMO Massively Multiplayer Online. 28

MOBA Multiplayer Online Battle Arena. 28

OS Operating System. 20, 41, 42

RAM Random-Access Memory. 22, 30, 64, 65

RDW Redirected Walking. 5–7

RPG Role Playing Game. 9, 28

SCIPT Selfcontained Inertial Position Tracking. 5

SDK Software Development Kit. 24, 26

SLAM Simultaneously Localize and Map. 6

TCP Transmission Control Protocol. 34, 35

TFT Thin-Film-Transistor. 17

UBT UnrealBuildTool. 54

UDP User Datagram Protocol. 34, 35, 47, 66, 69

UE4 Unreal Engine 4. ix, xi, 2, 3, 19, 23–25, 27–34, 37–39, 44, 47–50, 53, 54, 56, 59, 63–67, 69

UI User Interface. 10–13, 15, 20–22, 26–30, 41, 54

UMG Unreal Motion Graphics. 3, 23, 29, 32, 66, 69

UML Unified Modeling Language. 40, 44, 45, 49

USB Universal Serial Bus. 43

VE Virtual Environment. 1–3, 5–9, 12, 23, 25, 32–34

VM Virtual Machine. 20, 47

VoIP Voice over IP. 34

VR Virtual Reality. ix, xi, 1–3, 5–12, 23–26, 31–34, 38, 41, 43, 48, 54, 56, 59, 60, 63, 66, 69

WIM World-In-Miniature. 8

WLAN Wireless Local Area Network. ix, xi, 2, 3, 5, 6, 19, 24, 26, 27, 34, 44, 47, 56, 65, 66, 69, 72

XML Extensible Markup Language. 21, 42

Bibliography

[1] Android. https://developer.android.com/index.html. Accessed: 2016-08-11.

[2] Android Studio IDE. https://developer.android.com/studio/index.html. Accessed: 2016-08-11.

[3] Android Version Distribution. https://developer.android.com/about/dashboards/index.html. Accessed: 2016-08-11.

[4] Android Wear. https://www.android.com/wear/. Accessed: 2016-08-11.

[5] Android Wear - Exiting Full-Screen Activities. https://developer.android.com/training/wearables/ui/exit.html. Accessed: 2016-08-11.

[6] Android's Material Design Guidelines. https://developer.android.com/design/index.html. Accessed: 2016-08-11.

[7] Apple's iOS Guidelines. https://developer.apple.com/ios/human-interface-guidelines/. Accessed: 2016-08-11.

[8] Arm HUD VR Menu. http://blog.leapmotion.com/arm-hud-widget-like-smartwatch-entire-arm/. Accessed: 2016-08-11.

[9] Cryengine - Game Engine. https://www.cryengine.com/. Accessed: 2016-08-11.

[10] Hovercast VR Menu. http://blog.leapmotion.com/hovercast-vr-menu-power-fingertips/. Accessed: 2016-08-11.

[11] Leap Motion VR Best Practices Guidelines. https://developer.leapmotion.com/vr-best-practices. Accessed: 2016-08-11.

[12] MIT Technology Review Article on TheVoid. https://www.technologyreview.com/s/544096/inside-the-first-vr-theme-park/. Accessed: 2016-08-11.

[13] Nielsen Norman Group - Menu Design. https://www.nngroup.com/articles/menu-design/. Accessed: 2016-08-11.

[14] Oculus Rift User Interface Guidelines. https://developer.oculus.com/documentation/intro-vr/latest/concepts/bp_app_ui/. Accessed: 2016-08-11.

[15] OEM Off-Highway - Degrees Of Freedom. http://www.oemoffhighway.com/article/10979955/inertial-measurement-sensors-improve-safety-in-ag-equipment. Accessed: 2016-08-11.

[16] Project Soli. https://atap.google.com/soli/. Accessed: 2016-08-11.

[17] The Void. https://thevoid.com/. Accessed: 2016-08-11.

[18] Unity - Game Engine. https://unity3d.com/. Accessed: 2016-08-11.

[19] Unreal Engine 4 - Game Engine. https://www.unrealengine.com/. Accessed: 2016-08-11.

[20] Unreal Engine 4 Documentation. https://docs.unrealengine.com. Accessed: 2016-08-11.

[21] VR Menu Design 1. https://www.wearear.de/virtual-reality-menu-design-part1/. Accessed: 2016-08-11.

[22] Windows's Universal Windows Platform Design Guidelines. https://developer.microsoft.com/en-us/windows/design. Accessed: 2016-08-11.

[23] Worldwide Smartphone Sales. https://www.gartner.com/newsroom/id/3270418. Accessed: 2016-08-11.

[24] Worldwide Wearable Devices Sales. https://www.gartner.com/newsroom/id/3198018. Accessed: 2016-08-11.

[25] Eric R Bachmann, Michael Zmuda, James Calusdian, Xiaoping Yun, Eric Hodgson, and David Waller. Going anywhere anywhere: Creating a low cost portable immersive ve system. In *Computer Games (CGAMES), 2012 17th International Conference on*, pages 108–115. IEEE, 2012.

[26] Eric Badique, Marc Cavazza, Gudrun Klinker, Gordon Mair, Tony Sweeney, Daniel Thalmann, and Nadia M Thalmann. Entertainment applications of virtual environments. 2002.

[27] Mudit Ratana Bhalla and Anand Vardhan Bhalla. Comparative study of various touchscreen technologies. *International Journal of Computer Applications*, 6(8):12–18, 2010.

[28] Frank Biocca. Communication within virtual reality: Creating a space for research. *Journal of Communication*, 42(4):5–22, 1992.

[29] Jim Blascovich, Jack Loomis, Andrew C Beall, Kimberly R Swinth, Crystal L Hoyt, and Jeremy N Bailenson. Immersive virtual environment technology as a methodological tool for social psychology. *Psychological Inquiry*, 13(2):103–124, 2002.

[30] Doug A Bowman and Larry F Hodges. An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 35–ff. ACM, 1997.

[31] Doug A Bowman and Larry F Hodges. Formalizing the design, evaluation, and application of interaction techniques for immersive virtual environments. *Journal of Visual Languages & Computing*, 10(1):37–53, 1999.

[32] Doug A Bowman, Ernst Kruijff, Joseph J LaViola Jr, and Ivan Poupyrev. *3D user interfaces: theory and practice.* Addison-Wesley, 2004.

[33] Doug A Bowman and Chadwick A Wingrave. Design and evaluation of menu systems for immersive virtual environments. In *Virtual Reality, 2001. Proceedings. IEEE*, pages 149–156. IEEE, 2001.

[34] Sarah S Chance, Florence Gaunet, Andrew C Beall, and Jack M Loomis. Locomotion mode affects the updating of objects encountered during travel: The contribution of vestibular and proprioceptive inputs to path integration. *Presence*, 7(2):168–178, 1998.

[35] Roger M Downs and David Stea. *Cognitive maps and spatial behavior: Process and products.* na, 1973.

[36] Sascha Gebhardt, Sebastian Pick, Thomas Oster, Bernd Hentschel, and Torsten Kuhlen. An evaluation of a smart-phone-based menu system for immersive virtual environments. In *3D User Interfaces (3DUI), 2014 IEEE Symposium on*, pages 31–34. IEEE, 2014.

[37] Dominique Gerber and Dominique Bechmann. Design and evaluation of the ring menu in virtual environments. *Immersive projection technologies*, 2004.

[38] James Gosling. *The Java language specification.* Addison-Wesley Professional, 2000.

[39] JoAnn T Hackos and Janice Redish. User and task analysis for interface design. 1998.

[40] Eric Hodgson, Eric R Bachmann, David Vincent, Michael Zmuda, David Waller, and James Calusdian. Weavr: a self-contained and wearable immersive virtual environment simulation system. *Behavior research methods*, 47(1):296–307, 2015.

[41] Daniel A James, Neil Davey, and Tony Rice. An accelerometer based sensor platform for insitu elite athlete performance analysis. In *Sensors, 2004. Proceedings of IEEE*, pages 1373–1376. IEEE, 2004.

[42] Keiko Katsuragawa, Krzysztof Pietroszek, James R Wallace, and Edward Lank. Watchpoint: Freehand pointing with a smartwatch in a ubiquitous display environment. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 128–135. ACM, 2016.

[43] Hyung-il Kim and Woontack Woo. Smartwatch-assisted robust 6-dof hand tracker for object manipulation in hmd-based augmented reality. In *2016 IEEE Symposium on 3D User Interfaces (3DUI)*, pages 251–252. IEEE, 2016.

[44] Brenda Laurel and S Joy Mountford. *The art of human-computer interface design.* Addison-Wesley Longman Publishing Co., Inc., 1990.

[45] Rung-Huei Liang and Ming Ouhyoung. A real-time continuous gesture recognition system for sign language. In *Automatic Face and Gesture Recognition, 1998. Proceedings. Third IEEE International Conference on*, pages 558–567. IEEE, 1998.

[46] Soo-Chul Lim, Jungsoon Shin, Seung-Chan Kim, and Joonah Park. Expansion of smartwatch touch interface from touchscreen to around device interface using infrared line image sensors. *Sensors*, 15(7):16642–16653, 2015.

[47] Jennifer Mankoff and Gregory D Abowd. Cirrin: a word-level unistroke keyboard for pen input. In *Proceedings of the 11th annual ACM symposium on User interface software and technology*, pages 213–214. ACM, 1998.

[48] Merryn J Mathie, Adelle CF Coster, Nigel H Lovell, and Branko G Celler. Accelerometry: providing an integrated, practical method for long-term, ambulatory monitoring of human movement. *Physiological measurement*, 25(2):R1, 2004.

[49] Deborah J Mayhew. *Principles and guidelines in software user interface design.* Prentice-Hall, Inc., 1991.

[50] Scott McGlashan and Tomas Axling. A speech interface to virtual environments. In *Proc., International Workshop on Speech and Computers*, 1996.

[51] Stan Melax, Leonid Keselman, and Sterling Orsten. Dynamics based 3d skeletal hand tracking. In *Proceedings of Graphics Interface 2013*, pages 63–70. Canadian Information Processing Society, 2013.

[52] Rolf Molich and Jakob Nielsen. Improving a human-computer dialogue. *Communications of the ACM*, 33(3):338–348, 1990.

[53] Florian Müller, Sebastian Günther, Niloofar Dezfuli, Mohammadreza Khalilbeigi, and Max Mühlhäuser. Proxiwatch: Enhancing smartwatch interaction through proximity-based hand input. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pages 2617–2624. ACM, 2016.

[54] Tao Ni and Patrick Baudisch. Disappearing mobile devices. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 101–110. ACM, 2009.

[55] Donald A Norman. *The design of everyday things: Revised and expanded edition.* Basic books, 2013.

[56] Iana Podkosova, Khrystyna Vasylevska, Christian Schoenauer, Emanuel Vonach, Peter Fikar, Elisabeth Broneder, and Hannes Kaufmann. Immersivedeck: A large-scale wireless vr system for multiple users. In *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*. IEEE, March 2016.

[57] Ivan Poupyrev, Mark Billinghurst, Suzanne Weghorst, and Tadao Ichikawa. The go-go interaction technique: non-linear mapping for direct manipulation in vr. In *Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 79–80. ACM, 1996.

[58] Ivan Poupyrev, Numada Tomokazu, and Suzanne Weghorst. Virtual notepad: handwriting in immersive vr. In *Virtual Reality Annual International Symposium, 1998. Proceedings., IEEE 1998*, pages 126–132. IEEE, 1998.

[59] Sharif Razzaque, Zachariah Kohn, and Mary C Whitton. Redirected walking. In *Proceedings of EUROGRAPHICS*, volume 9, pages 105–106. Citeseer, 2001.

[60] Gerhard Reitmayr, Chris Chiu, Alexander Kusternig, Michael Kusternig, and Hannes Witzmann. iorb-unifying command and 3d input for mobile augmented reality. In *Proc. IEEE VR Workshop on New Directions in 3D User Interfaces*, pages 7–10, 2005.

[61] Daniel Rodríguez-Martín, Carlos Pérez-López, Albert Samà, Joan Cabestany, and Andreu Català. A wearable inertial measurement unit for long-term monitoring in the dependency care area. *Sensors*, 13(10):14079–14104, 2013.

[62] Robert J Seidel and Paul R Chatelier. *Virtual reality, training's future?: perspectives on virtual reality and related emerging technologies*, volume 6. Springer Science & Business Media, 2013.

[63] Sidney L Smith and Jane N Mosier. *Guidelines for designing user interface software.* Mitre Corporation Bedford, MA, 1986.

[64] Mads Soegaard and Rikke Friis Dam. *Encyclopedia of Human-Computer Interaction.*

[65] Richard Stoakley, Matthew J Conway, and Randy Pausch. Virtual reality on a wim: interactive worlds in miniature. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 265–272. ACM Press/Addison-Wesley Publishing Co., 1995.

[66] Bjarne Stroustrup. *The C++ programming language.* Pearson Education India, 1995.

[67] Richard P Troiano, David Berrigan, Kevin W Dodd, Louise C Masse, Timothy Tilert, Margaret McDowell, et al. Physical activity in the united states measured by accelerometer. *Medicine and science in sports and exercise*, 40(1):181, 2008.

[68] David Waller, Eric Bachmann, Eric Hodgson, and Andrew C Beall. The hive: A huge immersive virtual environment for research in spatial cognition. *Behavior Research Methods*, 39(4):835–843, 2007.

[69] Mary C Whitton, Joseph V Cohn, Jeff Feasel, Paul Zimmons, Sharif Razzaque, Sarah J Poulton, Brandi McLeod, and Frederick P Brooks. Comparing ve locomotion interfaces. In *IEEE Proceedings. VR 2005. Virtual Reality, 2005.*, pages 123–130. IEEE, 2005.

[70] William Winn, Hunter Hoffman, Ari Hollander, Kimberley Osberg, Howard Rose, and Patti Char. Student-built virtual environments. *Presence: Teleoperators and virtual environments*, 8(3):283–292, 1999.

[71] Robert Xiao, Gierad Laput, and Chris Harrison. Expanding the input expressivity of smartwatches with mechanical pan, twist, tilt and click. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 193–196. ACM, 2014.

[72] Xiaoping Yun, Eric R Bachmann, Hyatt Moore, and James Calusdian. Self-contained position tracking of human movement using small inertial/magnetic sensor modules. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 2526–2533. IEEE, 2007.

[73] Shumin Zhai and Paul Milgram. Human performance evaluation of manipulation schemes in virtual environments. In *Virtual Reality Annual International Symposium, 1993., 1993 IEEE*, pages 155–161. IEEE, 1993.