SAPIENZA
UNIVERSITÀ DI ROMA

Design, development and evaluation of a framework for recording and synchronizing experiences in VR with physiological signals

Facoltà di Ingegneria dell'informazione, informatica e statistica
Corso di Laurea Magistrale in Engineering in Computer Science

Candidate
Danilo Bernardini
ID number 1544247

Thesis Advisor
Prof. Massimo Mecella

Academic Year 2017/2018

Thesis not yet defended

Design, development and evaluation of a framework for recording and synchronizing experiences in VR with physiological signals
Master thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: danilo.bernardini93@gmail.com

Contents

Chapter 1

Project concept and technologies

This chapter introduces the project idea and the concepts behind it, starting from the use of VR combined with physiological signals and going towards use cases and project requirements. After this, involved technologies - virtual reality and physiological sensors - are illustrated in details and current state of the art is presented, including the available devices on the market. Finally, the last section is about the specific technologies used in this project.

1.1 VR and physiological signals

Virtual reality is a powerful tool not only for commercial use, but also for research. Since the beginning of its lifetime, in fact, people started to think that it could have been a good environment to test and study certain phenomenons, especially concerning human body and behavior. From the first VR basic prototypes to the advanced systems of these last years, researchers have conducted lost of experiments about people reactions and emotions with VR making use of physiological signals. The reason why virtual reality is so appropriate and useful for research is that it can simulate real situations, so it gives researchers countless opportunities: they can recreate real world environments, test new kinds of interaction, make people relive specific scenes, and so on.

As mentioned above, for many years there have been several examples of studies about VR with the use of physiological signals. An early research paper dealing with this subject was the one about fear of flying written by Wiederhold and others and published in 1998 [1]. The authors wanted to test if there are significant physiological differences between people who are afraid of flying and people who are not. In order to accomplish that, they studied the body responses of two groups of people, one suffering from a fear of flying and the other one not, while experiencing a virtual reality flight. They measured heart rate, peripheral skin temperature, respiration rate, sweat gland activity and brain wave activity. The results of the experiment were successful and they could conclude that there exist significant differences between these two groups of people. In addition, the authors also succeeded in reducing arousal of people with fear of flying making them experience the simulation for several times.

The following year another research paper about VR and physiological signals was published by Cobb and others [2]. This was one of the first investigations about VR sickness (see 1.3.3): the authors examined effects and symptoms of virtual reality on people using several methodologies, including physiological parameters.

These examples are part of the several research studies conducted to analyze or treat specific phobias and diseases. More recent ones are, for example, those by Kuriakose and Lahiri [3] and by Seinfeld and others [4], published in 2015 and 2016, respectively. Both are about anxiety: the former aims at measuring anxiety level in adolescents with autism using physiological signals while being in VR, the latter examines the influence of music on anxiety induced by fear of heights in VR.

Finally, we can include in this list a paper about the hand illusion in VR with temperature monitoring by Llobera and others [5] and one about reduction of pain and anxiety in dental procedures using VR distractions [6].

1.2 Idea and requirements

There are many experiments and studies using virtual reality that also include physiological signals; the previous section presented some of them showing that VR applications are limitless and that using physiology with it can lead to important results. There is a common aspect in all these kind of studies: if the researchers want to collect physiological data during a virtual reality experience, they need to set up everything. This means to find a way to follow the experiment, to see what the participant sees in VR, to store data and to be able to analyze it. All these things are possible only if the collected data - VR video, physiological signals and other available sources - are synchronized. This process implies a lot of effort from the researchers because they have to figure out how to design and realize it, wasting time they could invest in the actual research.

This leads to the basic motivation behind this thesis project: designing and implementing a framework for recording and synchronizing experiences in VR with physiological signals. The framework would be a useful tool for whoever wants to physiologically analyze a certain phenomenon in VR. Having a ready-to-use test environment would allow to save time and focus on the actual experiment; the only required thing is to build the virtual scene, everything else is done by the framework.

1.2.1 Use cases

The project requirements come from use cases. We can imagine two main cases that cover the two different parts of the application usage:

1. examiner wants to see in real-time and to record what the participant sees and how the body behaves;

2. examiner wants to replay a previously recorded session.

Let us analyze these points one by one in order to fully understand what the system should do. After a brief description, for each use case is presented a diagram followed by the list of steps required to arrive to the goal, including the extensions.

1. Since the framework is designed for conducting VR researches with physiological signals, who conducts the experiment should be able to see what the participant sees in the virtual world in real-time and how the body reacts to the experience. In addition to this, the examiner should be able to record the whole session including all the signals and streams he has. This leads to an extension of the case, named *Record session*, which can be executed only after the examiner is able to see everything correctly. Once recorded, everything can be synchronized in order to be easily accessible and analyzable in the future. This brings the *Record session* case to be extend in turn. The use case description assumes that the participant is ready, with all the physiological sensors attached to his body and the VR headset correctly placed.

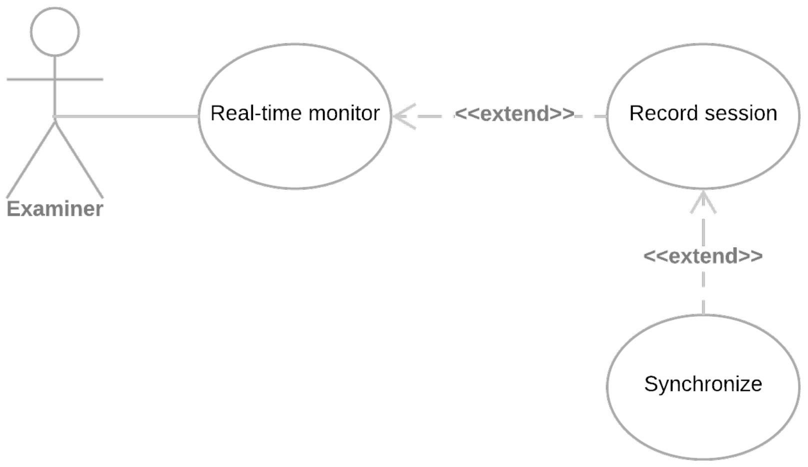


Figure 1.1. *Real-time monitor* case diagram

Name	Real-time monitor
Initiator	Examiner
Goal	Monitor the session in real-time

Main success scenario
1. Examiner runs the application
2. Examiner sets up the sensors in the application
3. Application displays sensors, VR and camera streams
4. Examiner sees everything in real-time

Extensions
4. Examiner wants to record the session
   a. Examiner starts a new recording
   b. VR, camera and sensors streams are stored
   c. Examiner stops the recording

4c. Examiner wants to synchronize the session
   a. Examiner starts the synchronization
   b. Synchronization completes

2. The examiner should have the possibility to offline replay a previously recorded session, being able to see everything as it was during the real-time monitoring. In this replay mode there should be the possibility to find key moments in which some relevant event happened. Since the session sources are synchronized, this task should be easy. The examiner should be able to tag these events placing markers on them: these would allow a simple navigation on the session and would also be very useful for analysis.

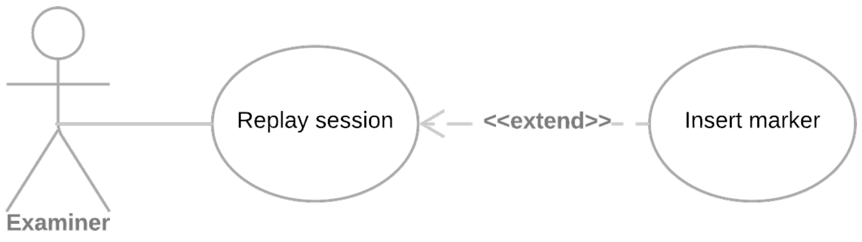


Figure 1.2. *Record session* case diagram

Name	Replay session
Initiator	Examiner
Goal	Replay a prerecorded session

Main success scenario
1. Examiner runs the application
2. Examiner selects the "replay" mode
3. Examiner opens one of the directories containing a previous recorded session
4. Examiner plays the session

Extensions
4. Examiner wants to insert one or more marker
   a. Examiner finds the moments he wants to mark
   b. Examiner inserts the marker and assigns it a label

The whole UML Use Case Diagram is represented in figure 3.3.

1.2.2 Requirements

This section is about the requirements for the project, which result from the above use cases and from some other considerations.

The system should allow to monitor a virtual reality experience while some sensors collect the participant's physiological signals. It would be reasonable to include in the system a camera that films the participant during the experience. This would make it possible to also have an external view on the user, being able to monitor his physical reactions to events. Examiners can see if the participant

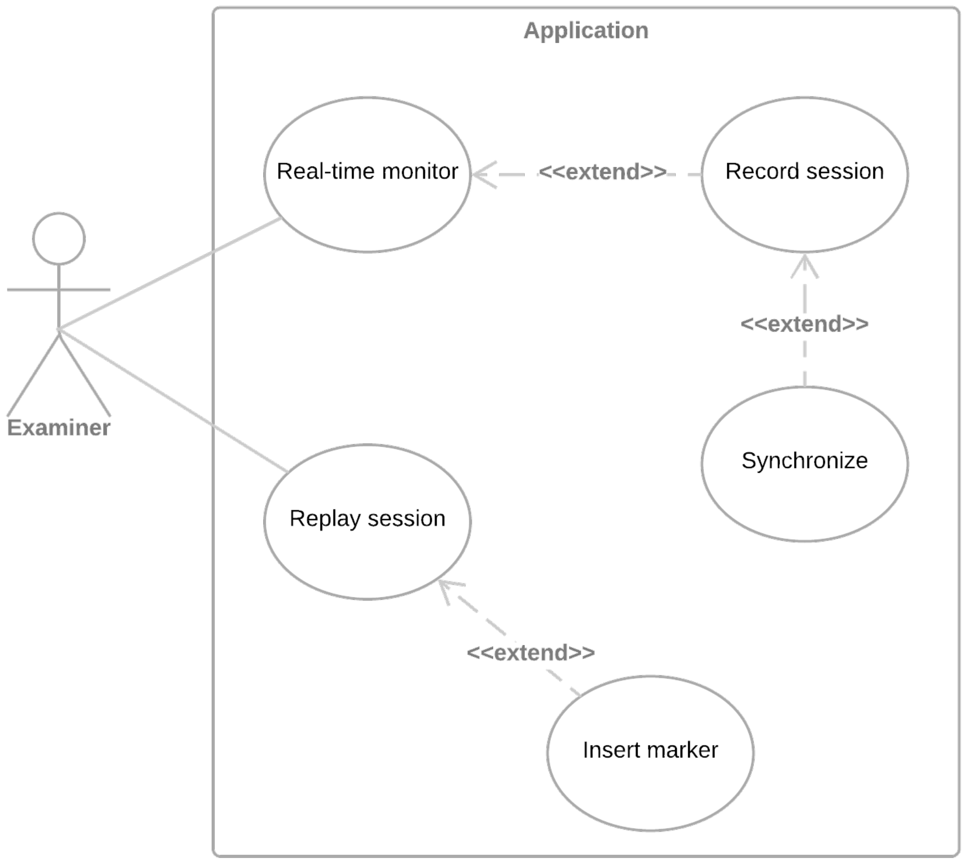


Figure 1.3. Global Use Case Diagram

moves, rotates his body, is not stable, or has any other kind of reaction in response to a virtual stimulus.

Moreover, system structure and usability should not be hard, considering that the potential users will not be engineers or computer science experts. These people, in fact, will reasonably be scientist, doctors or researchers that need to test and analyze something in VR.

The system should allow to to start a new real-time monitoring session which shows what the participant sees in VR, the camera source filming him and his physiological signals. During this "live" session, the examiner should be able to start a new recording: this will store the streams from VR, camera and sensors. At the end of the experiment the examiner should stop the recording and the various data need to be automatically synchronized. Alongside with the real-time mode, the system should have an "offline" part that allows to replay a previously recorded session. This mode should provide classical playback controls such as play, pause and stop, as well as timeline navigation to quickly skip to a specific frame. Since everything is already synchronized, in this modality the examiner can easily track every moment of the experiment; if there are relevant events it is worth to emphasize, the system should allow to place markers on them. Finally, the stored files (VR video, camera video and sensors data) should remain independent from each other, i.e. they should still be accessible from external programs after the synchronization and markers insertion.

1.3 Virtual Reality

Virtual Reality (VR) is a technology that aims to allow a user to experience a computer-generated simulated environment. It commonly consists of a visual experience, but it can also include audio, haptic, touch and other feedback devices.

Visual devices are usually VR headsets, head-mounted displays (HMD) with two screens - one per eye - that immerse the user in a virtual world simulating the real one. While wearing the headset, the user can look around rotating his head, move in the environment or interact with virtual objects using controllers, and perceive other senses through ad-hoc devices.

The above systems are usually computer-based, i.e. they are physically connected to the computer, so the applications run here and the visual output is transmitted to the headset display. Another kind of solution is the one that uses the smartphone: in this case the headset is just a case that holds the device, the whole computation takes place in the smartphone and thus the applications are much simpler and less realistic compared to the computer-based ones (see 1.3.4).

1.3.1 VR spread

Virtual reality idea is not something new, there have been basic examples of it since the 1960s. The first head-mounted display system was the *Sword of Damocles* [7] created in 1968 by computer scientist Ivan Sutherland and his student Bob Sproull: the graphics were very primitive and consisted only of wireframe rooms, but the device was able to change the showed perspective according to the user head position. Its name comes from its appearance: the whole system was too heavy to be worn by a person so it was attached to the ceiling, suspended on the user's head.

In the next years different VR systems were developed, most of the times for the videogames industry. Examples can be found in some *SEGA* and *Nintendo* products.

The main reason why VR is exploded in the last few years is that technology is now powerful enough to support advanced computation and to give users really immersive experiences. Until a few years ago VR solutions could not satisfy *immersion* requirements (see section 1.3.3): graphics were not good enough and they had low refresh rates, causing the user some sickness (see section 1.3.3).

Conversely, in the present days we can rely on powerful machines and tools that allow us to create and experience good simulations of the world, most of the times without perceiving any issue.

VR spread is also encouraged by the release of not very expensive headsets (see section 1.3.4): a lot of people can now buy VR systems, either for development or entertainment, but also companies are starting to use them for business (see next section).

We can refer to *Gartner Hype Cycle for Emerging Technologies* of 2017 [8] to understand VR position in the market. Gartner is an American research and advisory company that provides IT-related insights and market analysis. The Hype Cycle is "a graphical depiction of a common pattern that arises with each new technology or other innovation" [9], i.e. a chart that shows the maturity and social application of a technology. The plot is divided in five phases representing the stages of a technology life cycle: Innovation Trigger, Peak of Inflated Expectations, Trough of Disillusionment, Slope of Enlightenment, Plateau of Productivity. Gartner releases it every year and in 2017 they stated that transparently immersive experiences are one of the 3 most trending topics, together with AI everywhere and digital platforms, and that VR is currently in the fourth phase of the cycle. This means that VR applications are starting to be understood and adopted by companies, that methodologies and best practices are being developed and that the technology is beginning to spread to the potential audience.

1.3.2 Applications

Virtual Reality has many applications in different fields, from gaming and entertainment to education and healthcare. Here are some examples of industries that are using VR for improving and enhancing their work.

Business: Companies can make clients experience virtual tours of business environments; they can test and show new products before releasing them or train their employees using VR.

Culture and education: VR can be used in museums and historical settings to recreate ancient sites, monuments, cities; it can be very useful for teaching, since students can be immersed in what they are studying and better understand things (e.g. history, geography, astronomy).

Games and entertainment: Gaming and entertaining industry is probably the biggest adopter of VR, since this brings the player in a whole new level of immersion and realism; VR is also used in movies, sport and arts.

Healthcare: VR allows to simulate a human body, so that students and doctors can study and train on it; it can simulate a surgery or perform a robotic one, i.e. control a robotic arm remotely; it can be used to treat phobias and diseases.

Military: VR can be used to perform a combat simulation where soldiers can train and learn battlefield tactics; it can be also useful for flight simulation or medical training on the battlefield.

Science and engineering: Virtual reality technology can help scientists to visualize and interact with complex structures, or it can be used by engineers to design, model and see something in 3D.

1.3.3 Measures

The previous sections mentioned the words *immersion*, *realism*, *sickness*, etc. These are concepts we can analyze and measure during VR experiences.

Immersion and presence

There is a lot of confusion about the concepts of immersion and presence; some people think they are the same thing, some others mix them up. According to Mel Slater, immersion concerns *what the technology delivers from an objective point of view. The more that a system delivers displays (in all sensory modalities) and tracking that preserves fidelity in relation to their equivalent real-world sensory modalities, the more that it is 'immersive'* [10]. This means that immersion is something that can be determined and somehow measured in an objective way, depending entirely on the technology and not on the user. Presence, on the contrary, is a user-dependent concept, something that varies depending on the person, a human reaction to immersion: each person can perceive a different level of presence inside the same system and even a single user can perceive a different level of presence using the same system in different times, depending on the user's emotional state, past history, etc. [11]. We can say that presence is both immersion and involvement: this regards the state of focus and attention of the user and depends on the interaction with the virtual world, the storyline, and other similar features of the experience.

Measuring immersion Since immersion is an objective concept depending only on technology, we can measure it evaluating how close the system video, audio and other features are to the real world [11]. For example if we consider visual source, we can identify some parameters that influence it:

- field of view and field of regard, respectively the size of the visual field that can be viewed instantaneously and the total size of the visual field surrounding the user;
- display size and resolution;
- stereoscopy and head-based rendering, given by head tracking;
- frame rate and refresh rate.

Measuring presence Presence is a subjective measure and is not easy to quantify. During the years some techniques have been developed and tested, the most used ones are the following [12]:

Questionnaires Users participate to the virtual experience and then answer a questionnaire about presence. The questions imply responses between two extremes (e.g. from "no presence" to "complete presence"). The problem with this approach is that asking questions about presence can affect the actual perception of the participant.

Behavioral We can see evidence of presence if users in the virtual world behave as if they were in the real world. This can be triggered by events that cause a physical reaction on the participant, such as a movement reflex or a body rotation.

Physiological signals This is a specialization of the previous approach: if we know how a person physiologically reacts to an event, and we can find the same reaction during a virtual event, then this is a sign of presence. This technique can only be used when we have well-known and easy to measure reactions, for example fear, so it is not ideal for calm and "boring" scenarios where nothing happens.

Breaks in presence (BIP) A break in presence is an event that occurs when the user becomes aware he is in a virtual environment. This can happen if the visual stimuli starts to lag, if the graphics become low-quality, if the user touches a physical object from outside the VR experience, etc. This approach allows to know presence by analyzing the moments when BIP occur, and it is a good alternative to the physiological one because it can be used in every kind of environment (calm, stressfull,scary, and so on).

Co-presence

When more than one participant share the same virtual environment we can measure co-presence (also known as shared presence). It is the feeling that the other participants are really present and that the user is interacting with real people [13]. As well as presence, also co-presence is measured with questionnaires. It is not proven that co-presence is related to presence, because some studies [14, 15] seem to find correlation between them and others [13] do not.

VR sickness

As anticipated, virtual reality can cause some sort of sickness to the user. The common symptoms are similar to motion sickness symptoms: general discomfort, nausea, sweating, headache, disorientation, fatigue [2]. Sickness varies from person to person and it is usually caused by conflicts between perceived movement and actual movement: if the player walks in VR the eyes say he is walking while the ears do not detect any movement, creating confusion to the brain. Other aspects that can induce sickness are low refresh rate and poor animations: both these things have the same effect on brain, which processes frames at higher rates or expects better animations.

Since it is a subjective feeling, the most common way of measuring VR sickness is through questionnaires. The standard methodology for measuring sickness is the *Simulator Sickness Questionnaire (SSQ)* by Kennedy, Lane, Berbaum and Lilienthal [16]. An interesting alternative approach is to monitor the postural activity of the participant [17]: it seems that motion sickness is related to postural stability and that there are differences in postural activity between people who are experiencing sickness and people who are not.

Situation awareness

A general measure that applies also for VR is situation awareness. It consists in having the awareness of what is happening in the surrounding environment and what may happen in the future, being ready to handle the situation that will arise. A famous approach to measure it is the *Situational Awareness Rating Technique (SART)* [18] originally developed by Taylor and Selcon in 1990 for evaluating pilots. It is a post-trial questionnaire that asks the user to rate 10 dimensions with a number from 1 to 7.

Workload

Another measure that can be useful in VR is workload, meant as the effort needed to complete a task. The most used technique is the *NASA Task Load Index (NASA-TLX)* [19], a questionnaire divided in two parts: the first one consists of 6 rating subscales, the second one is a personal weighting of these subscales.

1.3.4 VR headsets

This section presents the most important available VR devices, starting from the computer-connected (tethered) headsets and continuing with mobile ones. Finally, standalone VR devices are introduced.

Tethered VR headsets

Computer-connected VR systems take advantage of the computing power of the machine they are connected to, so they can give the user complex environments and experiences. The headsets provide head tracking and motion tracking - generally through external base stations - so the user can move with 6 degrees of freedom (DOF) and can interact with the virtual world in a lot of possible ways.

HTC Vive HTC Vive system consists of a headset, two controllers and two base stations to track body movements. The display has 90 Hz refresh rate and 110 degree field of view, with a resolution of 1080x1200 per eye. The base stations also track the controllers, allowing an advanced interaction with the virtual environment. Together with several sensors, the headset also includes a front-facing camera. In 2018 HTC launched the Pro version of the Vive, fitted with a higher-resolution display, attachable headphones and a second camera.

Oculus Rift Oculus initiated a Kickstarter campaign in 2012 and in 2014 it was acquired by Facebook. The system includes two Touch controllers and the headset hardware is basically the same as the HTC Vive's one. Tracking is obtained with *Constellation*, an optical-based tracking system that detects IR LED markers on the HMD and controllers.

Sony PlayStation VR PlayStation VR is Sony's proprietary VR system only compatible with PlayStation 4. It supports only PS4 ad-hoc games but there is the possibility to play any other PS4 game like if it was on a very large screen. Headset display has a resolution of 960x1080 per eye, a 100 degrees field of view and a native refresh rate of 90 or 120 Hz. Regarding the controller input, the system supports both regular *DualShock 4* or *PlayStation Move* controllers. Players need a *PlayStation Camera* to track the HMD and the Move controllers.

Mobile VR headsets

A mobile VR system consists of a case that holds the smartphone and two lens that separate the display in two parts, one per eye. Since the only available sensors are those included in the smartphone, these VR systems can not track body movements and therefore they can just count on 3 DOF (head rotation). Since they are generally simpler and less powerful than the tethered ones, mobile VR systems are usually quite cheaper.

Google Daydream View Daydream is the enhanced successor of the *Google Cardboard*, a very basic cardboard-made headset with two lenses that turns the smartphone into a VR system. Daydream software is built into the Android operating system since the Nougat version. The package comes with a wireless touchpad controller that can be tracked with on-board sensors.

Samsung Gear VR Gear VR is a system only compatible with some high-level Samsung devices, it contains a controller equipped with a touchpad and a motion sensor. The development was carried out by Samsung in collaboration with Oculus, which took care of the software distribution.

Standalone VR headsets

Standalone headsets are a new type of VR systems that does not require a computer or a smartphone in order to work. Since they have almost the same technical specifications, these devices computing power can be compared to that of smartphones, even if they are optimized for VR applications.

Oculus Go This device was developed by Oculus in collaboration with Qualcomm and Xiaomi. The HMD has 3 degree of freedom and is equipped with a 5.5-inch display with a resolution of 1280x1440 per eye, a Snapdragon 821 processor and comes with a 32GB or 64GB storage. The system also includes a wireless controller.

Lenovo Mirage Solo Mirage Solo is a brand new device and it works with Google Daydream platform. Technically speaking, its display is similar to the Oculus Go's one but the device is powered by a Snapdragon 835 with a 4GB RAM and it has a better tracking system, that gives the headset 6 DOF (but only 3 for the controller).

1.3.5 Tracking devices

Together with headsets, a lot of different sensors and devices can be used to enhance VR experience. The most significant improvement in interaction is probably tracking. Most HMDs have an integrated head tracking system that allows 6 DOF movement, and controllers are usually tracked making it possible to interact with the environment. However, there exist external systems that enable advanced tracking functionalities such as full body tracking, hand tracking or eye tracking.

Body tracking

Body tracking makes possible to follow movements of the whole body, included arms and legs. It is a technique often used in movies and videogames to animate digital characters in CGI (in this case it is known as *motion capture*). There are several companies providing body tracking solutions, here follows a list of some of the most valuable products currently available.

HoloSuit It is a suit that allows full body motion tracking and capturing thanks to the many sensors it has embedded. It also provides haptic feedback and it is compatible with the most important operating systems and development environments (*Unity*, *Unreal Engine*).

HTC Vive Trackers Trackers are small disc-shaped devices that work with HTC Vive and they can be attached to any real physical object in order to track its movements. They can be also connected to the user to obtain body tracking or to replace Vive controllers.

Optitrack Optitrack is one of the largest motion capture companies in the world, providing powerful tools and devices able to track small markers from long distances.

Perception Neurons This system is composed of small units called Neurons that are placed to the various parts of the body, for example to fully track a person 11 to 32 Neurons are required. It is compatible with most 3D modeling and animation programs.

PrioVR PrioVR uses sensors attached to key points of the body to provide full tracking without the need of a camera. It comes with small controllers and it works with both Unity and Unreal Engine.

Orbbec Orbbec produces long-rage depth cameras that can be used with the proprietary body tracking SDK to fully detect the human body. It works with Windows, Linux and Android.

Senso Suit Senso Suit is a kit composed of 15 modules, each containing a sensor and a vibrating motor, that make it possible to achieve full body tracking. SDK available for Unity, Unreal Engine, C++ and Android.

VicoVR VicoVR is a wireless device that provides full body tracking to mobile VR headsets without the need of body-attached sensors. The device look is similar to *Microsoft Kinect* (discontinued) and is compatible with Android, iOS and the main mobile VR headsets.

Xsens Another big name in tracking industry, Xsens provides a variety of solutions for full body tracking, including suits or strap-based sensors kits. It does not need a camera and works with almost every 3D animation program.

Hand tracking

Hand tracking is a complex aspect of artificial intelligence that allows to detect and track hand movements only with a camera, without the need of any controller or external device. Here are presented two devices and a software library capable of detecting hand movements.

LeapMotion LeapMotion tracking system is a camera that needs to be attached to a VR headset in order to track hands with a field of view of 135 degrees. It works with both HTC Vive and Oculus Rift.

uSens Fingo Fingo consists of a camera similar to LeapMotion that works with both tethered and mobile VR headsets, but it is optimized for mobile. Currently it is only compatible with Unity.

OpenCV OpenCV (Open source Computer Vision) is a software library that provides several computer vision features, including hand tracking. It is written in C++ but there exist wrappers for other programming languages.

Eye tracking

Eye tracking is the process of detecting eye position and gaze. It is usually implemented with lights and cameras that analyze eye movement and detect the direction of gaze. Eye tracking is used in a lot of different fields such as safety (for example in a car), advertising, marketing and psychology.

aGlass aGlass is an eye tracking module compatible with HTC Vive. It is composed of two lenses that are placed above the Vive lenses and provide low-latency eye tracking with a field of view of 110 degrees.

Fove Fove is a VR headset with an incorporated eye tracking module. It is compatible with SteamVR and OSVR and supports both Unity and Unreal Engine.

Pupil Pupil Labs produces binocular eye tracking add-ons for the leading VR headsets. It comes with open source software and it works with Unity.

Tobii Tobii proposes an hardware eye tracking development kit for HTC Vive or a retrofitted version of the Vive with an eye tracking integration.

1.4 Physiological signals

Physiological parameters are vital signals that describe a person's functions and activities state. There exist many different physiological signals, measurable with a lot of different sensors and devices. This section presents a wide variety of physiological sensors, from cheap wearable devices to expensive professional kits.

1.4.1 Wearable devices

Empatica E4 Wristband Wearable band and app for visualization and analysis, it comes with mobile API and Android SDK.

Sensors: photoplethysmogram (PPG), accelerometer, EDA, thermometer.

EQ02 LifeMonitor Device that can be worn on the chest and that stores or transmits the data to a mobile phone or a computer.

Sensors: ECG, respiration, skin temperature, accelerometer.

Helo LX Smartband and app, it is compatible with Windows, Android, iOS.

Sensors: heart rate, breath rate, blood pressure.

Microsoft Band 2 Smartband by Microsoft, it can also work with Cortana. The app is available for Android, iOS and Windows Phone.

Sensors: heart rate

Polar H10 Heart rate sensor to attach on the chest, it comes with an internal memory and it also works with third-party applications.

Sensors: heart rate

Xiaomi Mi Band 2 Band mainly used to monitor fitness exercises and to track sleep activity.

Sensors: accelerometer, heart rate

1.4.2 Professional kits

Biopack Systems Hardware and software platform that provides professional measurements and analysis for research and education. The main module can acquire up to 16 different channels but it has no internal memory and it needs an external power supply.

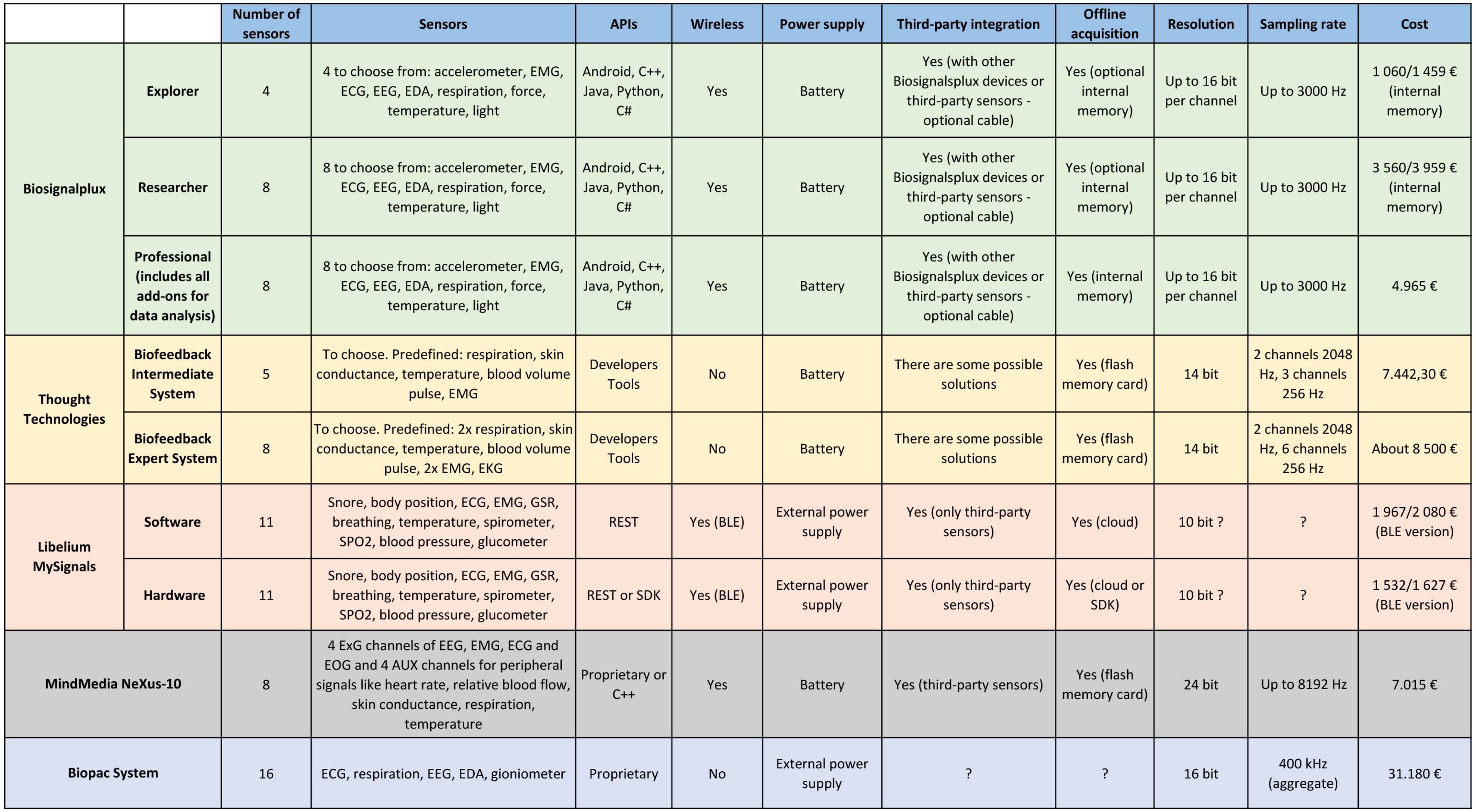
BiosignalsPlux BiosignalsPlux makes research kits that include 4 or 8 sensors and a 4 or 8 channels wireless hub, which has internal memory and battery. It works with third-party applications and sensors and comes with Android, C++, Python, Java and MATLAB APIs.

Libelium MySignals Software (uses an includes box screen) or hardware (uses Arduino) versions, acquired data is sent to the cloud and can be visualized on the web or on the mobile app. Up to 18 different sensors can be connected to the system. Available SDK for the hardware version.

MindMedia NeXus-10 Wireless acquisition system that supports 8 different signals, it includes a software for visualization and synchronization, together with a SD card and a battery for ambulant recordings.

Thought Technology Biofeedback System 5 or 8 channels system, it comes with proprietary software for data visualization and includes a memory card for offline acquisitions.

Figure 2.4 shows a complete comparison between these professional kits.



		Number of sensors	Sensors	APIs	Wireless	Power supply	Third-party integration	Offline acquisition	Resolution	Sampling rate	Cost
Biosignalplux	Explorer	4	4 to choose from: accelerometer, EMG, ECG, EEG, EDA, respiration, force, temperature, light	Android, C++, Java, Python, C#	Yes	Battery	Yes (with other Biosignalsplux devices or third-party sensors - optional cable)	Yes (optional internal memory)	Up to 16 bit per channel	Up to 3000 Hz	1 060/1 459 € (internal memory)
	Researcher	8	8 to choose from: accelerometer, EMG, ECG, EEG, EDA, respiration, force, temperature, light	Android, C++, Java, Python, C#	Yes	Battery	Yes (with other Biosignalsplux devices or third-party sensors - optional cable)	Yes (optional internal memory)	Up to 16 bit per channel	Up to 3000 Hz	3 560/3 959 € (internal memory)
	Professional (includes all add-ons for data analysis)	8	8 to choose from: accelerometer, EMG, ECG, EEG, EDA, respiration, force, temperature, light	Android, C++, Java, Python, C#	Yes	Battery	Yes (with other Biosignalsplux devices or third-party sensors - optional cable)	Yes (internal memory)	Up to 16 bit per channel	Up to 3000 Hz	4.965 €
Thought Technologies	Biofeedback Intermediate System	5	To choose. Predefined: respiration, skin conductance, temperature, blood volume pulse, EMG	Developers Tools	No	Battery	There are some possible solutions	Yes (flash memory card)	14 bit	2 channels 2048 Hz, 3 channels 256 Hz	7.442,30 €
	Biofeedback Expert System	8	To choose. Predefined: 2x respiration, skin conductance, temperature, blood volume pulse, 2x EMG, EKG	Developers Tools	No	Battery	There are some possible solutions	Yes (flash memory card)	14 bit	2 channels 2048 Hz, 6 channels 256 Hz	About 8 500 €
Libelium MySignals	Software	11	Snore, body position, ECG, EMG, GSR, breathing, temperature, spirometer, SPO2, blood pressure, glucometer	REST	Yes (BLE)	External power supply	Yes (only third-party sensors)	Yes (cloud)	10 bit ?	?	1 967/2 080 € (BLE version)
	Hardware	11	Snore, body position, ECG, EMG, GSR, breathing, temperature, spirometer, SPO2, blood pressure, glucometer	REST or SDK	Yes (BLE)	External power supply	Yes (only third-party sensors)	Yes (cloud or SDK)	10 bit ?	?	1 532/1 627 € (BLE version)
MindMedia NeXus-10		8	4 ExG channels of EEG, EMG, ECG and EOG and 4 AUX channels for peripheral signals like heart rate, relative blood flow, skin conductance, respiration, temperature	Proprietary or C++	Yes	Battery	Yes (third-party sensors)	Yes (flash memory card)	24 bit	Up to 8192 Hz	7.015 €
Biopac System		16	ECG, respiration, EEG, EDA, gioniometer	Proprietary	No	External power supply	?	?	16 bit	400 kHz (aggregate)	31.180 €

Figure 1.4. Professional physiological kits comparison

1.5 Used technologies

This section presents the technologies used in the project and explains the reasons why they were chosen.

Virtual reality

The framework is meant to be used on a desktop environment, so for what concerns virtual reality the only possible choices were the Oculus Rift and the HTC Vive. The latter was the chosen one for the project. The main reason behind this decision was that it allows the user to physically move better than the Oculus, considering that its base stations can track a room-scale environment. Since the application can be used in a variety of different projects and experiments, this choice was the best for a general-purpose framework.

For what concerns the development environment of the virtual experience, the chosen tool was *Unity3D*, a game engine available for Windows and MacOS that also supports virtual reality. Its scripting language is C#, it is easy to use, compatible with HTC Vive and the created games can be launched in stand-alone mode, so it was the best choice for this project.

Programming language

The application will handle the majority of the logic of the framework (see section 2.2). It needs to be able to play videos easily and with the regular control commands (play, pause, stop), it has to support signals graphs representation and the development phase should not cost too much effort. From these considerations we can conclude that the best programming language to use is Java. There are a lot of useful external libraries that can be imported from it, it has ready-to-use media player classes, nice graphical interfaces tools and it is cross-platform.

Physiological sensors

The last choice to make is about which sensors to use in the framework. There were several aspects that were taken into account in order to decide which professional kit was the best among those reported in figure 2.4. The kit should support the larger possible number of different sensors; it should be wireless, so that it is easier to connect and more practical to use; it should have internal memory and a battery, so that it can be also used for other kind of applications (e.g. external experiments); it should be compatible with third-party hardware or software and provide APIs in order to be usable with custom services and applications. The only option that satisfied all these condition was the BiosignalsPlux Researcher kit: 8 sensors, wireless, portable and with well documented API for several programming languages. The chosen sensors were: ECG, EMG, EEG, EDA, respiration, accelerometer (one channel per axis), temperature and force.

Chapter 2

Design

This chapter describes the design of the system, starting from the initial idea and continuing with the multiple steps until the final scheme.

2.1 Similar systems

Before thinking about the design itself, a research on similar projects was conducted in order to better understand how this kind of systems are realized. The list includes not only projects that are close to this one as a whole, but also projects that provide just some similar features. Here follows the research result, divided in the various fields regarding the functionalities of the system.

PhysioVR [20] is a framework developed to integrate physiological signals in mobile VR applications. Signals are acquired using wearable devices connected to the smartphone via Bluetooth and include heart rate, electroencephalography (EEG) and electromyography (EMG). The system allows to record data and to communicate with the VR content allowing an external event triggering using a real-time control.

MuLES [21] is a tool for acquiring and streaming EEG signals that aims at facilitate the development of brain-computer interface programs. The system allows to replay a session and there are clients developed for use on different platforms and in various programming languages.

PhysSigTK [22] is a toolkit for accessing low-cost physiological signals hardware from Unity. It aims at helping developers to exploit engagement in their effective games.

RehabNet [23] is a system that helps patients rehabilitation through serious videogames that monitor physiological parameters. It supports a large variety of external physiological sensors.

Bicycle4CaRe [24] is a tool for domestic physical training that uses VR and sensors to monitor user's activity. It simulates a home environment and collects data both from it and from the user, providing real-time feedback.

Cube [25] is a VR simulation platform for the supervision of personnel working in critical infrastructures. It collects physiological data during the virtual experience to identify the optimal physiological profile of personnel for using it in real scenarios.

Athena [26] is an analytics platform that acquires data from a combat simulator, combining this data with physiological signals of the player, and sends him body feedback.

VAST [3] is a virtual reality system for social communication that also collects physiological signals to determine the user's anxiety level.

VR-SAAFE [27] is a VR-based system developed for understanding facial emotions in subjects suffering of schizophrenia. The system uses an eye tracker and physiological signals to monitor user's eye gaze and emotions.

2.1.1 Design

The above systems use different approaches for what concerns design.

- *PhysioVR* is composed of two layers: the first one deals with synchronizing the devices and streaming the data, the second one is a Unity package that receives and analyzes data and physiological signals. There are also other two modules for external control and data recording.
- *MuLES* acts as the server part of a client-server architecture. It handles the communication between the several EEG sensors and transmits their data to the client applications. The devices are connected with their own drivers and the connection with the clients is realized through TCP/IP.
- *RehabNet* architecture is based on three building blocks: an hardware layer that deals with devices connection, a control panel that filters and cleans data, and a web interface for accessing the rehabilitation tools. The whole system is organized as a client-server application.
- *Bicycle4CaRe* is composed of different biofeedback devices connected to the PC via an Arduino board,ca desktop application and a VR simulated environment. The app contains the main functionalities and allows to control the experience.
- *Athena* is consists of a data acquisition layer that collects the signals from the sensors and the game data from the combat simulator. This layer sends data to IBM's Infosphere Streams, that processes it and generates analytics for storage.
- *VAST* is divided in three modules. The first one presents the tasks, the second one adapts and changes tasks according to the users behavior and the last one is the data acquisition layer.
- *VR-SAAFE* is divided in three components too: task presentation environment based on Unity, eye tracking application and the physiological signals acquisition module.

2.1.2 Physiological sensors

This section presents the physiological sensors used by similar projects. This could help understanding which are the most used and useful signals in this kind of applications. *PhysioVR* uses only three signals:

- heart rate, acquired through Android wearable devices such as smartbands, smartwatches or camera-based sensors;
- EEG, obtained from a low-cost -wearable headband called Muse BCI;
- EMG, acquired with Myo Armband, a device that contains 8 sensors and that recognizes gestures.

PhysSigTK uses sensors considering criteria such as ease of use, price, software support and low-level access to data. Therefore the employed devices are four, measuring several different signals:

- Empatica E3/E4, that measures EDA, blood volume, temperature and movement;
- Wild Divine Iom, that provides skin conductance level (SCL) and heart rate;
- e-Healt, an Arduino-based device that supports different sensors. In this case heart rate, galvanic skin response (a.k.a. EDA), ECG and breathing activity;
- NeuroSky Mindwave, a low-cost EEG sensor that also provides a heart rate sensor for the ear.

For what concerns *RehabNet*, it uses protocols such as the *Virtual-Reality Peripheral Network (VRPN)* and *Open Sound Control (OSC)* for integrating a large variety of external devices and sensors (trackers, haptic devices, analog input, sound, etc). In addition to this, it natively supports three sensors:

- EEG acquired with Emotiv EPOC, a wireless EEG headset;
- EMG, collected through Myomo mPower 1000, a prosthetic arm that also integrates sensors;
- kinematic data, acquired with the Microsoft Kinect.

VR-SAAFE monitors physiological signals and eye gaze:

- Biopac Bionomadix measures PPG (from which heart rate can be extracted), skin temperature, EMG, respiration and galvanic skin response;
- Tobii X120 for eye tracking.

Bicycle4CaRe measures both user's parameters and environmental conditions. The latter are not relevant for the project of this thesis; the former are:

- heart rate acquired with a pulse-oximeter sensor;
- breath rate measured through a respiration monitor belt;
- blood pressure collected with a sphygmomanometer.

2.1.3 Connectivity and libraries

The project application will need to connect to the devices and to acquire signals. In order to be able to do it, similar systems techniques and methodologies were explored. Here are reported some examples of connectivity and external libraries approaches.

- *PhysioVR* transmits data over the UDP protocol. In this way the communication can be multicast and the transmission of all the physiological signals is done using a single port. Data can be accessed even from external devices with UDP capabilities. Users can access from remote and interact with the system bidirectionally, thus receiving data and sending feedback. The application also provides client scripts for different programming languages such as Java, C#, Python, MATLAB.
- *MuLES* acquires data from the hardware using third-party APIs, SDKs, drivers or implementations of the specific device communication protocol. On the other hand, transmission to the client application is realized through the TCP/IP protocol, which allows to send data to a local network or over the Internet. This means that client scripts can be written in any programming language supporting basic socket programming, such as Java or Python.
- *RehabNet* sends data to a smartphone running the app using a custom implementation of the UDP protocol. It also connects to an analysis and tracking tool. As anticipated above, the system makes use of VRPN and OSC protocols in order to connect to other external devices or libraries, for example the OpenViBE BCI software.
- The interesting thing about *Cube* is that it provides different connectivity solutions that can be used simultaneously or not. An example of it is ECG acquisition: it can be wired or wireless (Bluetooth) transmitted to a local machine, transmitted to a smartphone over the Internet or using a satellite connection

2.1.4 Events and triggers

Since the system should deal with key events and markers, here are presented the solutions adopted by *PhysioVR* and *Bicycle4Care*.

- *PhysioVR* acquires data from the UDP connection and makes it available inside the Unity environment. This brings developers to have physiological data usable inside the experience, allowing some game parameters to be bound to physiological signals, or events to be triggered when certain conditions are verified. Another way of creating events is to have external inputs: the VR user can receive triggers influencing the experience scenario from an external application that monitors in real-time the person's vitals.
- *Bicycle4Care* reads data in real-time and guides the user through the experience with ad-hoc messages. Like for PhysioVR, events can be triggered when some parameter exceed a certain threshold.

2.1.5 Synchronization

After the acquisition, data needs to be synchronized. Among the presented systems, only *Athena* and *VAST* use synchronization or explicitly write about it.

- *Athena* acquires data from the combat simulator and the sensors and then it synchronizes them. This data is composed of physiological signals, game events like firing and target hits and haptic device activations. The adopted synchronization technique is not mentioned.
- *VAST* acquires real-time physiological signals and VR data in a synchronized manner for offline analysis. Markers are used to synchronize the different signals and the virtual environment.

2.1.6 Extensions

This section is not related with similar systems but presents two ideas for extension and improvement of VR experiences that can be useful for the project development.

- The first idea comes from S. Otsuka and others, who think that physiological signals could be easier acquired if the sensors are embedded in a VR headset [28]. They propose a new HMD that contains some sensors: their first prototype consists of a VR headset with an integrated PPG sensor that can measure heart rate. This system was tested and gave positive results, so this kind of devices could be a real and interesting thing for the future of VR and physiological signals research.
- The second one is about VR and eye tracking. A paper about this is the one by Pradeep Raj and others [29]. It presents the design of a VR-based social communication platform integrated with eye tracking technology. The authors believe that this kind of technology can be useful for presenting and analyze social situations and communication.

2.2 Design

After analyzing similar systems with their potentially useful features, now we can introduce the actual design of this thesis project. Since there have been problems and changes during the development phase, the general design has been modified during the implementation of the system. The development was divided into three prototypes, each adding a new functionality to the system:

1. real-time visualization of the VR video, the camera video and the signals stream;
2. recording and synchronization of the streams;
3. replay of the streams and insertion of key markers.

Here are presented all the design prototypes, from the initial idea to the final result.

2.2.1 Initial design

General architecture

The system architecture follows the actors involved in the process: the user and the examiner. The participant is the person who takes part in the experiment: he is equipped with a VR headset (HTC Vive), physiological sensors (BiosignalsPlux kit) and a webcam pointing to his face. The communication between these sensors and the examiner's computer is realized through controllers. The examiner component is constituted by the controllers, a recording and synchronization layer that saves the data on the storage and a Java application used to see in real-time what the user is experiencing and to start/stop the recording. Concerning the offline mode of the system, it is represented by two additional blocks. The first one deals with the offline part of the application itself, while the other represents the external applications that can be used to analyze data (e.g. MATLAB or Python). This last part is not covered by the project. Figure 4.1 shows an overall view of the system.

Controllers

Regarding the controllers, in the initial design they communicate with the Java application, that executes every task. In this case the application represents both the real-time and offline Java applications showed in figure 4.1. Starting from VR, the development platform is Unity and to send the video stream to the Java application we are going to use the *RockVR* plugin, which allows to record a video of what the user sees through the headset and send it to a streaming server in real time. The application is then going to read the stream and play it using the *MediaPlayer* Java class. For the physiological sensors we can use the official *Plux* API, which allows to easily acquire the signals data from Java. Finally, for the webcam we are going to use the *JavaCV* library, a wrapper of the *OpenCV library*, for visualizing and recording the video stream. For a detailed description of these tools and libraries see section 3.2. The controllers schema is represented in figure 4.2.

2.2.2 First prototype

Original schema

The first prototype only concerns the real-time visualization of the VR video, the camera video and the signals stream. Therefore, we can focus on the left side of figure 4.1: we only consider the user, the controllers and the real-time visualization application (figure 4.3). For what concerns the controllers, we are interest only in the real-time tasks, so the diagram is the one in figure 4.4.

Implementation changes

The original diagrams changed when implementing the prototype. The main problem was about the VR video streaming: encoding it, sending it to the streaming server (localhost) and decoding it generated a delay of about 3 seconds. This resulted in an out of sync visualization, since camera and physiological signals were

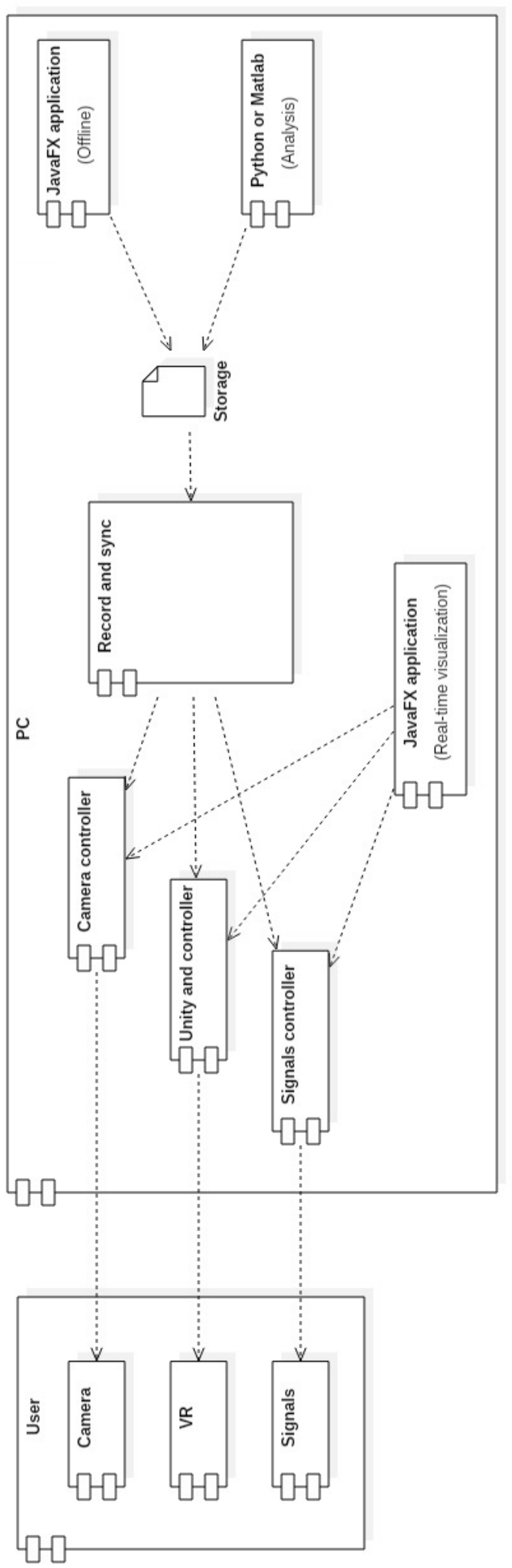


Figure 2.1. Initial general architecture

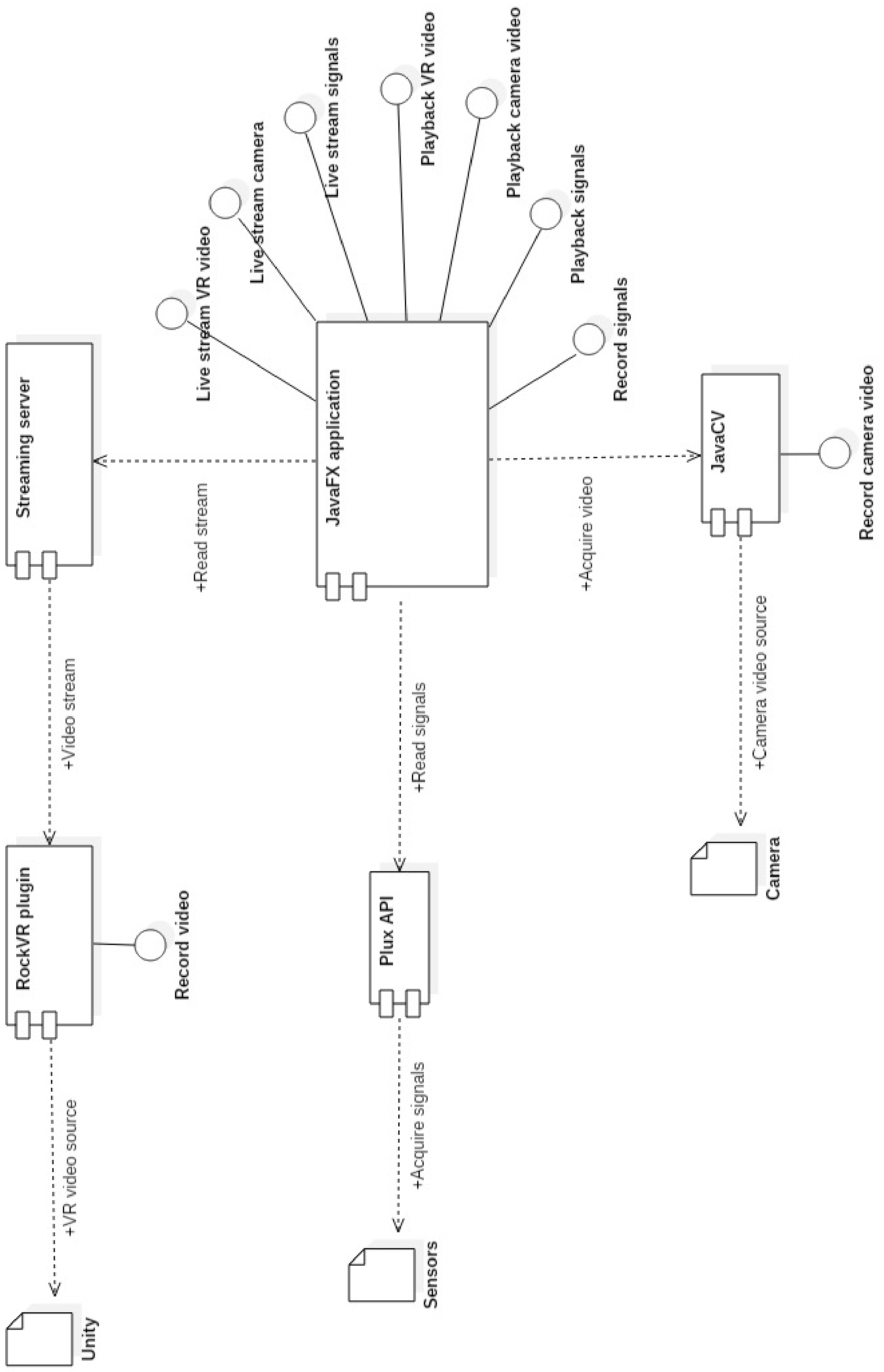


Figure 2.2. Controllers initial schema

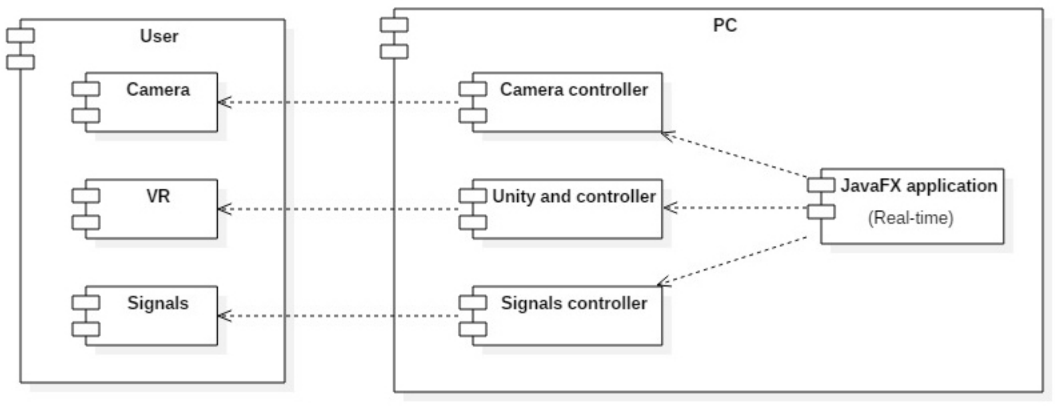


Figure 2.3. First prototype original architecture

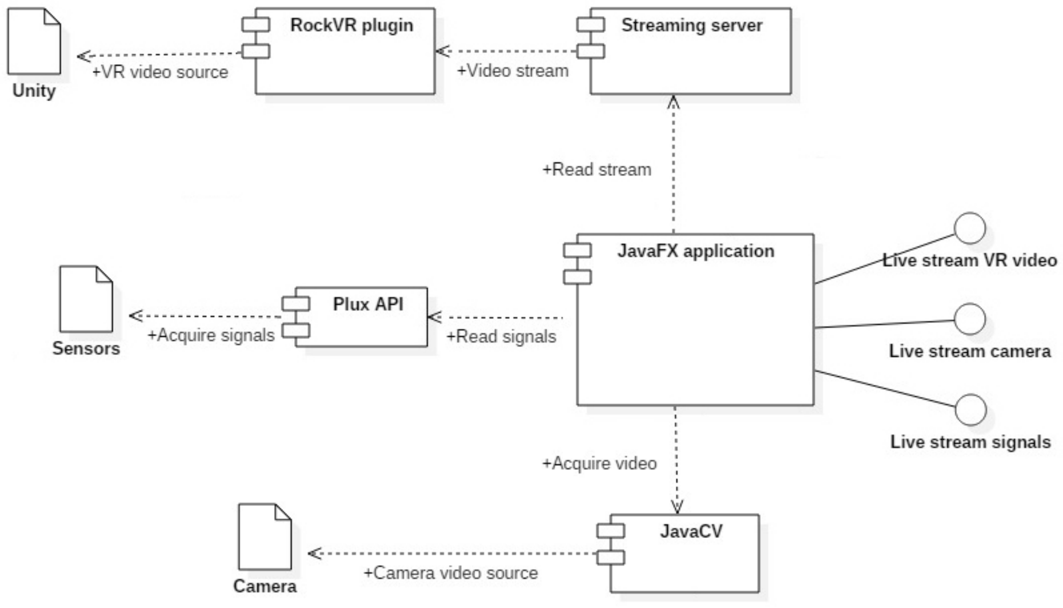


Figure 2.4. First prototype controllers original schema

synchronized but VR was not. This behavior was not acceptable. Even if it could have been possible to synchronize the three sources after the recording, it was not possible to see everything in real-time, failing to satisfy the first requirement.

Trying to find a solution to this problem was not easy. Playing the VR video stream from the Java application means to export what happens in Unity and to read it from an external application. The only suitable technique found was the above one. With it not working, the only reasonable solution was to use two screens and split the real-time part of the application in two pieces: one handling camera and signals visualization and one the VR visualization. The former will be handled by the Java application itself on the first screen and the latter by Unity on the second screen (figure 4.5).

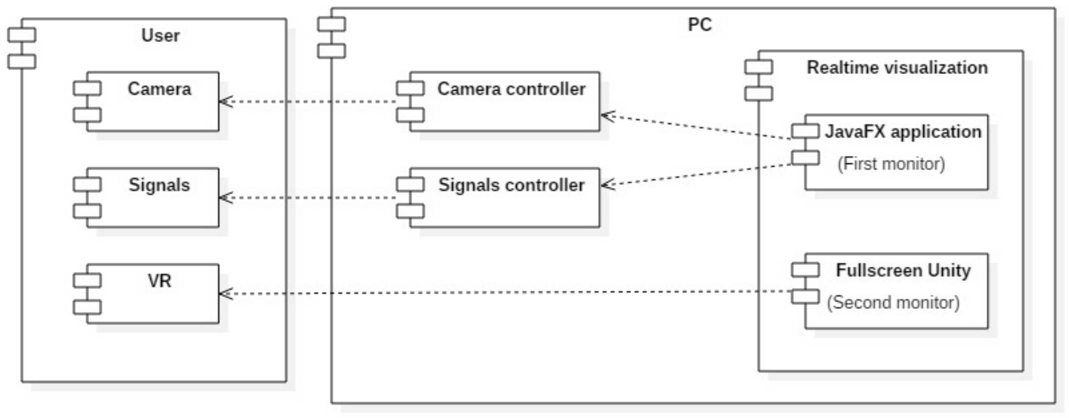


Figure 2.5. First prototype architecture implementation

As regards the controllers, we can simply get rid of the VR part because it has no more connections with the main application (figure 4.6).

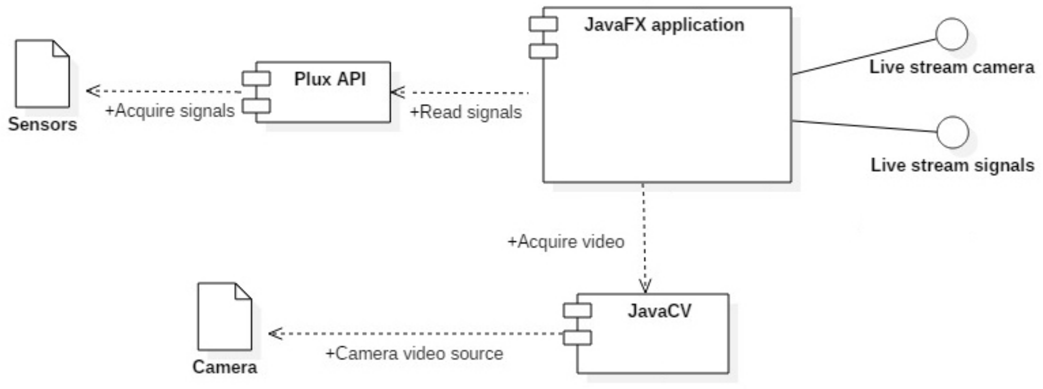


Figure 2.6. First prototype controllers implementation

2.2.3 Second prototype

The second prototype was supposed to add the recording functionality to the system. The general schema is exactly the same as the one in figure 4.5, but with the addition of the *Record and sync* module and of the storage (figure 4.7).

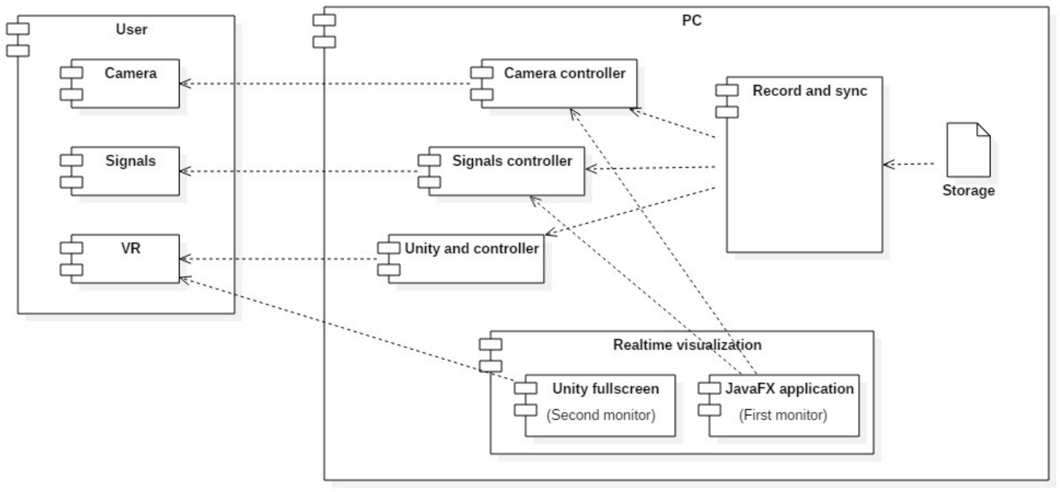


Figure 2.7. Second prototype architecture

Since from the first prototype architecture change we can not visualize the VR video in real-time from the Java application, we can not record it either, so we need to do it from Unity itself. This is the reason why there is also a Unity controller in the diagram (figure 4.8): this component is the RockVR plugin, and it will handle the recording of what the user sees and store it on the hard drive.

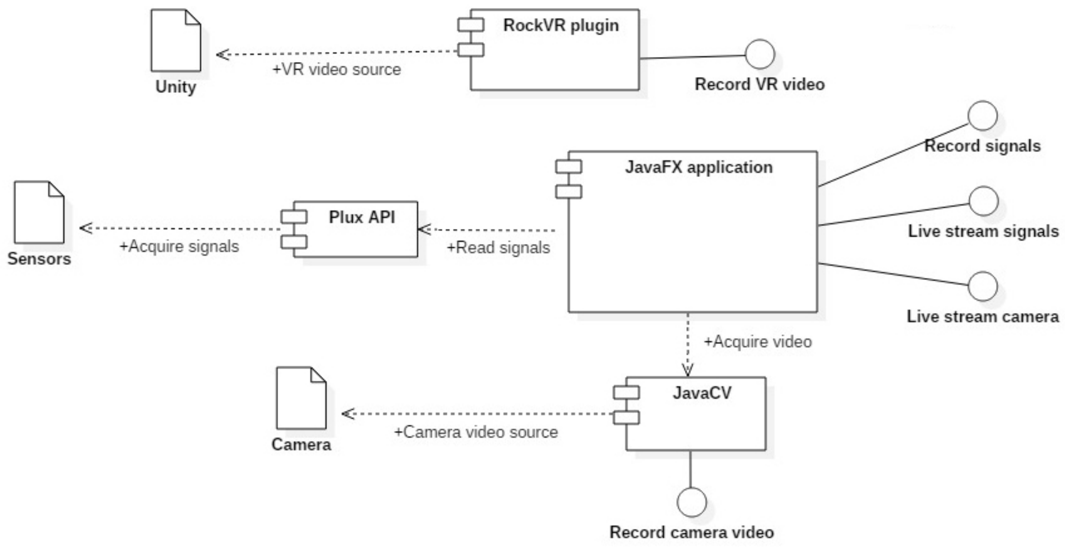


Figure 2.8. Second prototype controllers

This new consideration brings the system to be composed of two modules: the first one is a Java application that plays and records the camera and signals streams, the second one is a Unity package that records the VR video. At the end of the acquisition, the 3 files – VR video, camera video and signals data – are synchronized (see section 2.3) in order to be replayed and analyzed properly.

2.2.4 Third prototype

The third and last prototype represents the final version of the system. It adds the replay feature to the application, included the option to insert markers on specific moments of the playback.

Both the architecture and the controllers schemas are similar to the initial idea one, except for the separation of Java and Unity blocks (figure 4.9 and 4.10).

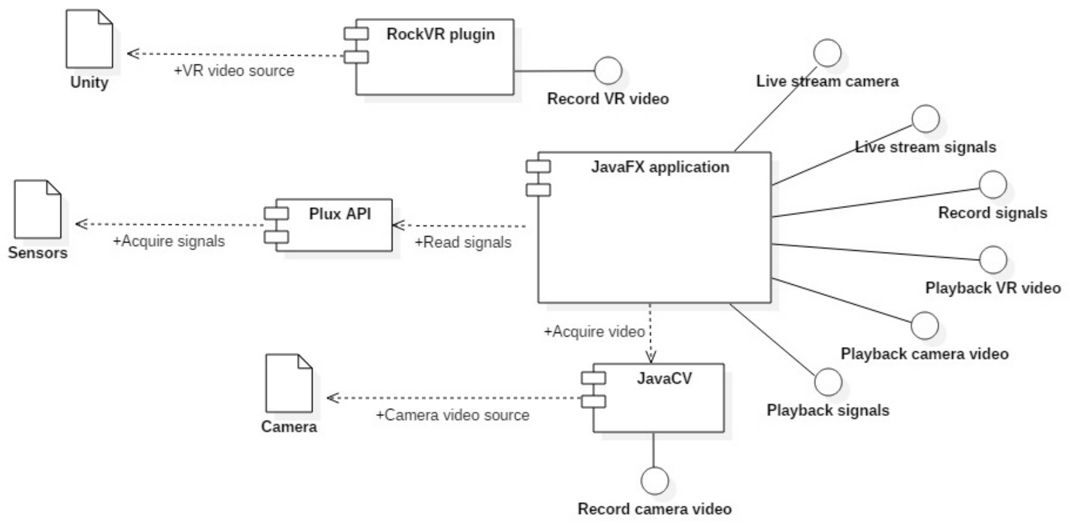


Figure 2.9. Third prototype controllers

2.3 Synchronization

Synchronization can be considered as one of the most important parts of this project. The value of the framework is that it gives researchers the possibility to have a ready-to-use environment that allows them to save valuable time, since everything they record can be easily synchronized. This section gives an overview of the synchronization problem and describes the solution adopted to solve it.

Synchronization issue

The camera video and the signals are acquired from the same application. Since the recording starts when the examiner clicks a button, both the sources start being stored approximately at the same time. There should be no relevant delays (i.e. more than one or two tenths of a second), therefore they should be already

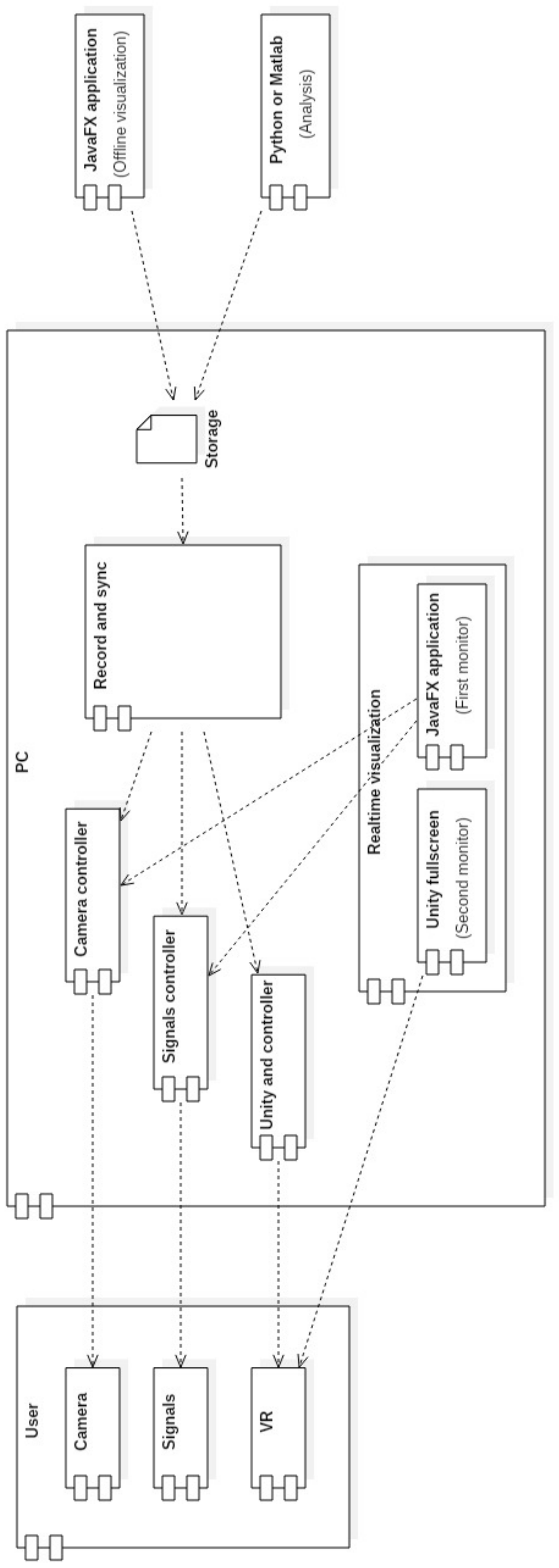


Figure 2.10. Third prototype architecture

synchronized. This means that if the implementation is done correctly, the camera video and the signals file do not need further processing in order to be synchronized.

The problem arises when we also consider VR video. As stated above, it is displayed on the Unity window itself and is recorded through a plugin. Just as the other recording, also this one is initiated by the examiner: since the two recordings events are generated by a human actor, there is no way they start at the exact same time.

We can consider the following as an example of a real usage of the platform. The examiner has already run the application and Unity, configured everything and started seeing the streams in real-time. Now he wants to record the whole session. He first starts recording from the Java application, acquiring signals and camera video, and then switches to the second screen and starts recording from Unity. Even if this two actions are executed very quickly, there will be an unavoidable delay (at least half a second) between the two recordings. This fact is not acceptable with respect to the framework idea and requirements, thus we need to find a way to really synchronize the files. Synchronization means to align the files or to cut them in order to display at the same moment events and data that happened at the same time. The ideal solution would have been to find a way of starting the two recordings simultaneously, without the double intervention of the examiner. This would have required some kind of connection between Java and Unity and was not easy to achieve. The solution adopted to address the problem was another: employ system timestamps.

System timestamps

Timestamps are sequences of characters representing a time unit, generally indicating a date and a time. In this case the used timestamps are the *Unix timestamps*, referring to the total number of seconds elapsed since midnight, January 1, 1970 UTC. When retrieved from a computer, this value depends on the underlying operating system. OS in fact measure time with different milliseconds granularity, for example Windows used 55 milliseconds until the 98 version and 10 milliseconds after.

The general idea about how to use timestamps for synchronization is to associate to each recorded file the timestamp, in milliseconds, of the moment when the recording starts. This gives us the indication about the time when the recordings started and therefore we can compute the difference between the values to get the exact delay between a source an another. The same steps are applied to the end of the recordings.

An overview on the synchronization phase can be described as follows. Without loss of generality we can assume that the examiner starts (and stops) recording from the Java application and then continues with Unity:

1. examiner starts the recording from the Java application;
2. current timestamp is associated to signals and camera video files as *initial timestamp*;
3. signals and camera video streams begin to be saved on the storage;

4. examiner starts the recording from Unity;
5. current timestamp is associated to VR video file as *initial timestamp*;
6. VR video stream begins to be saved on the storage;
7. examiner stops the recording from the Java application;
8. current timestamp is associated to signals and camera video files as *final timestamp*;
9. signals and camera video streams stop to be saved on the storage;
10. examiner stops the recording from Unity;
11. current timestamp is associated to VR video file as *final timestamp*;
12. VR video stream stops to be saved on the storage;
13. difference between signals (same as camera) and VR initial timestamps is computed (we can name this value);
14. difference between signals (same as camera) and VR final timestamps is computed (we can name this value *delayEnd*);
15. first *delayStart* milliseconds of signals and camera video files are cut;
16. last *delayEnd* milliseconds of VR video files are cut.

This procedure guarantees a good level of synchronization, which basically relies on timestamps accuracy. All the information about source files and timestamps is stored on simple text files. Implementation details about synchronization are reported in section 3.1.3.

Chapter 3

Realization

After the complete design of the system, it is time to talk about the actual implementation. This chapter explains some realization choices and gives details about core parts of the application.

3.1 Main development phases

Despite Java is a cross-platform language, Unity is only available for Windows and MacOS, and the former is more widespread that the latter. That is why the chosen operating system for the development and deployment of the framework was Windows. The development tool for the application was *NetBeans*, an integrated development environment (IDE) originally released for Java, but then expanded to support also other programming languages such as C, C++ and Javascript. Since the application needs to make use of external libraries, the best option was to use the build automation tool *Apache Maven*. It allows to easily describe dependencies and integrate libraries and plugins in the project, downloading them from repositories.

3.1.1 GUI

For what concerns the graphical interface of the application, the selected tool was *JavaFX*, a software platform for creating desktop applications. It aims at being the replacement of Swing as the standard GUI library for Java applications. Latest versions of JavaFX use *FXML*, an XML-based markup language, for defining user interfaces. The platform also provides a scene builder to create scenes by dragging and dropping components into it; this information is saved as FXML code, whose variables can be referred from Java controllers using the annotation *@FXML*.

The graphical interface structure is very basic and simple, providing only the really necessary components. The application is divided in two parts named and *Offline mode*, each with its own user interface. Their layouts are very similar, the only big difference is that the former contains signals configuration controls while the latter also includes VR video playback. The general idea is to have the screen horizontally split in half. The bottom part shows the signals while the upper part is in turn divided in two slices, one for the camera and one for the signals configuration (or VR video playback). Figure 4.1 shows an empty *Real-time mode* GUI.

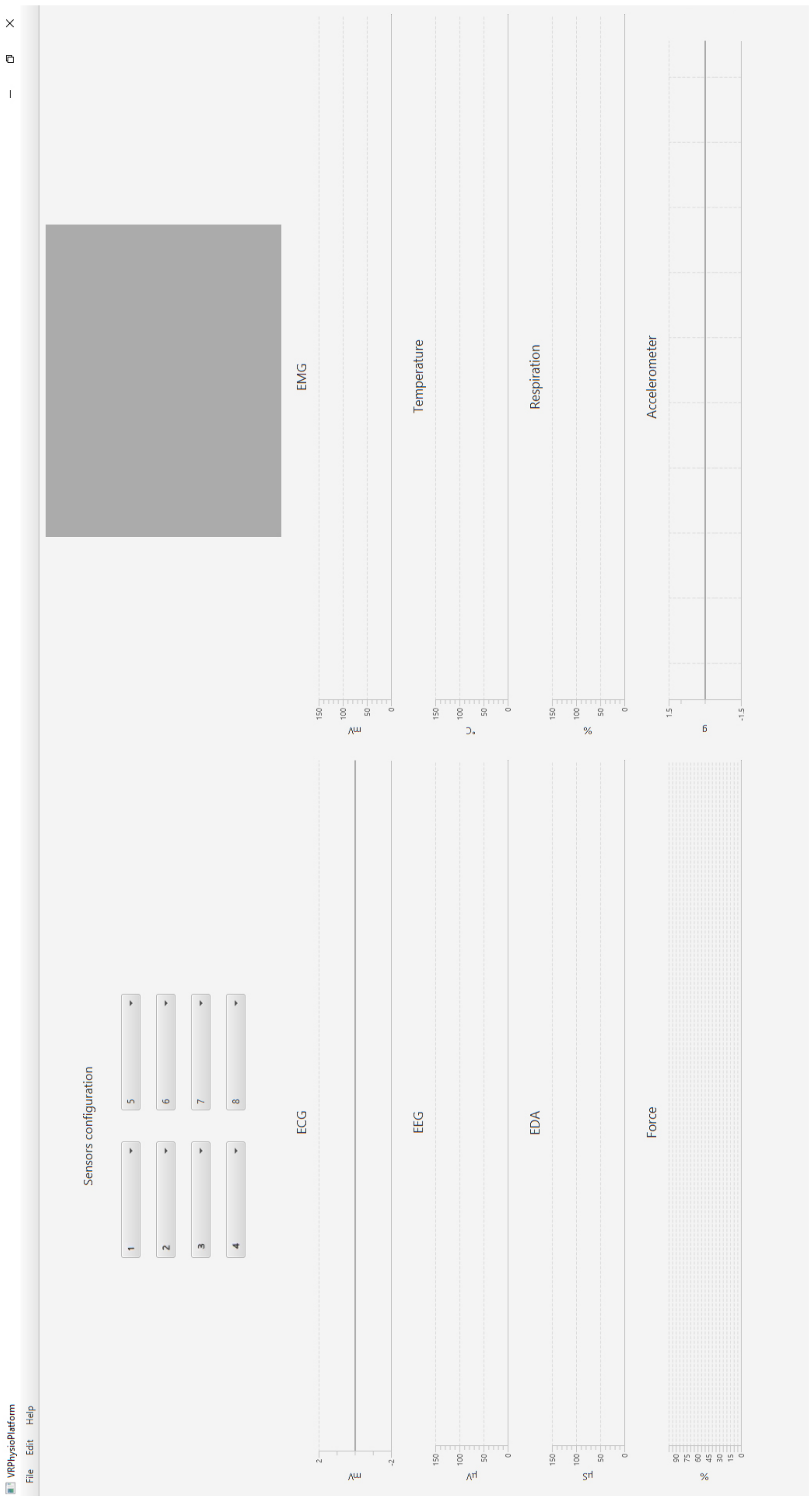


Figure 3.1. Real-time mode user interface

3.1.2 First prototype

The first prototype just had to show signals data and camera and VR videos in real-time. As already explained in the *Design* chapter, the real-time part pf the framework is divided in two pieces, one is the Java application, handling signals and camera video, and the other is Unity, showing the VR video. For what concerns Unity we do not need any development for this prototype, considering that we are only interested in seeing what the participant sees and Unity already does it in its predefined settings. When the virtual scene starts, it can be played in full-screen mode so that it is easy for the examiner to follow the participant's movements.

Regarding the other application, in this phase it has to be able to display camera video and physiological signals in real-time. Let us start with the camera.

Camera

The camera used during the whole development and testing of the system was a *Logitech* HD webcam connected to the computer via USB. As anticipated above, the software controller that makes camera accessible from Java is the *JavaCV* library (see section 3.2). It provides several methods for capturing and recording camera frames.

Going into implementation details of this feature, *OpenCVFrameGrabber* class is used to deal with camera input frames. First we need to create a new instance of the class passing as an argument the code of the camera. If the HTC Vive is also connected to the computer we have two active cameras, so the best thing to do is to give 0 as a parameter: this will make the system use the default camera or show a dialog to the user, who can select the one he wants. Next step is to call the *start()* method on the created instance. This causes the object to grab frames from the camera until the *stop()* command is called. In order to handle the incoming stream without blocking the whole execution we need to dedicate a new thread to it. This thread will execute a *Runnable* that captures and displays a frame placing it into an *ImageView*. First the grabber grabs a frame, then this frame is converted into an image that can be set into the ImageView.

When dealing with a graphical interface, every update of it should be executed on the JavaFX Application Thread. To solve this issue, the *setImage(image)* method was called inside a runnable passed as an argument to *Platform.runLater(Runnable runnable)*. This method makes sure that the specified runnable gets executed on the main thread at some unspecified time in the future.

In order to obtain a video containing 30 frames per second (FPS), the grabbing code was executed once every 33 milliseconds. This result was achieved thanks to the usage of the *ScheduledExecutorService* class. Code snippet 4.1 shows in details the described procedure.

```java
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import org.bytedeco.javacv.Frame;
import org.bytedeco.javacv.FrameRecorder;
import org.bytedeco.javacv.FrameGrabber;
import org.bytedeco.javacv.FFmpegFrameRecorder;
import org.bytedeco.javacv.Java2DFrameConverter;
import org.bytedeco.javacv.OpenCVFrameGrabber;
...
private ScheduledExecutorService cameraTimer;
private OpenCVFrameGrabber grabber;
private Java2DFrameConverter converter;
@FXML
private ImageView cameraView;
...

grabber = new OpenCVFrameGrabber(0);
converter = new Java2DFrameConverter();

try {
   grabber.start();
} catch (FrameGrabber.Exception ex) {
   logger.log(Level.SEVERE, "Error while initiating frame grabber", ex);
}

cameraTimer = Executors.newSingleThreadScheduledExecutor();
cameraTimer.scheduleAtFixedRate(new CameraRunnable(), 0, 33,
    TimeUnit.MILLISECONDS);

...

class CameraRunnable implements Runnable {

    @Override
    public void run() {
        Frame capturedFrame = null;
        try {
            capturedFrame = grabber.grab();
        } catch (FrameGrabber.Exception ex) {
            logger.log(Level.SEVERE, "Error while grabbing frame", ex);
        }

        Image image = SwingFXUtils.toFXImage
            (converter.convert(capturedFrame), null);

        Platform.runLater(new Runnable() {
            @Override
            public void run() {
                cameraView.setImage(image);
            }
        });
    }
}
```

Code snippet 3.1. Camera input management

Physiological signals

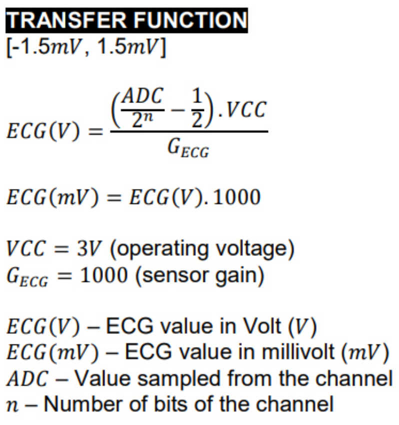
The other feature we need to implement for this prototype is signals data visualization. BiosignalsPlux device is connected to the computer wireless, using an included Bluetooth dongle. The communication between the device and Java is realized through the official Plux Java API (see section 3.2). This consists of a collection of basic classes and methods useful for retrieving sensors signals. The main class is *Device*, which represents the Plux device; the *Frame* class models a frame of acquired data and consists of a sequence number (byte), a boolean indicating the state of the digital input and the actual data. This is represented as an array with 8 elements (16 bits each), one per channel. Before starting the acquisition we need to find the connected device with the *Device.FindDevices(Vector v)*, which stores in *v* the MAC addresses of the discovered devices. Once found, a new Device object is created and the acquisition is started. Just as camera stream, also for signals we need a new thread to acquire data without freezing the whole application. Since Plux device reads data with a sampling rate of 1 kHz, (i.e. 1000 data frames per second), the thread is scheduled to be executed once every millisecond and it simply gets one single frame, storing it in a *Frame* object.

```java
import java.util.Vector;
import plux.newdriver.bioplux.*;
...
private String deviceMAC;
private Device dev;
private Device.Frame[] frames;
private ScheduledExecutorService signalsTimer;
...
Vector devs = new Vector();

try {
  Device.FindDevices(devs);

  if (devs.isEmpty()){
    Alert alert = new Alert(AlertType.ERROR, "Impossible to connect to
        the sensors. Please verify the connection and try again.");
    alert.setTitle("Sensors not found");
    alert.showAndWait();
   }
   else {
    deviceMAC = (String) devs.get(0);
    dev = new Device(deviceMAC);
    frames = new Device.Frame[1];
    frames[0] = new Device.Frame();
    dev.BeginAcq(1000, 0xFF, 16); // 1 kHz, 8 channels, 16 bits
   }
}
catch (BPException err){
  logger.log(Level.SEVERE, "Error while initializing the device", err);
}
```

```
if (!devs.isEmpty()){
   ...
   Thread getSignals = new Thread(() -> {
      try {
         dev.GetFrames(1, frames);
         ...
      } catch (BPException ex) {
         logger.log(Level.SEVERE, "Error while getting the frames", ex);
      }
   });

   signalsTimer = Executors.newSingleThreadScheduledExecutor();
   signalsTimer.scheduleAtFixedRate(getSignals, 0, 1,
       TimeUnit.MILLISECONDS);
}
```

Code snippet 3.2. Sensors data acquisition

Once acquired, signals data need to be visualized. JavaFX provides chart components that can be used to plot the data. For this kind of signals the best plot type is the line chart; we assign a chart to each signal source. The corresponding Java class is *LineChart<X, Y>*, where X and Y are the axis to be used. In this case we use a number containing the time value as X and a number indicating the sensor signal as Y. The easiest way of populating a chart is to add a *XYChart.Series<X, Y>* to it. This class is used to insert actual data to the chart. A series is assigned to the corresponding chart only when the examiner sets it in the *Sensors configuration* panel; the reason behind this behavior is explained in section 3.1.3. Once added to the chart, the series need to be regularly updated so that they can show the signals stream. The class that is commonly used in this kind of scenarios is *Timeline*, that allows to define an animation for a JavaFX object. The sensors provide 1000 frames per second but they are too many for visualization, so we can choose to only read some values. A good trade-off is to display 10 frames per second, i.e. to update the chart every 100 milliseconds: this is the value we set for the animation. Every time the animation is executed, the current data frame is read and the series is updated, therefore updating also the chart.



TRANSFER FUNCTION

[-1.5mV, 1.5mV]

$$ECG(V) = \frac{\left(\frac{ADC}{2^n} - \frac{1}{2}\right).VCC}{G_{ECG}}$$

$$ECG(mV) = ECG(V).1000$$

VCC = 3V (operating voltage)
G_{ECG} = 1000 (sensor gain)

$ECG(V)$ – ECG value in Volt (V)
$ECG(mV)$ – ECG value in millivolt (mV)
ADC – Value sampled from the channel
n – Number of bits of the channel

Figure 3.2. ECG sensor transfer function

Sensors acquire data with a 16 bits resolution, meaning that raw data consists of values ranging from 0 to 65335. To transform this numbers into meaningful measure values we need transfer functions. BiosignalsPlux provides datasheets containing information for each sensor including transfer functions. For example for the ECG sensor the function transforms the raw data into a value ranging from -1.5 mV to 1.5 mV. Figure 4.2 shows the ECG sensor transfer function. Functions were easily implemented in Java and used to plot the signals charts. Code snippet 4.3 shows the main parts of this phase, taking as an example the ECG chart; every other chart is visualized with the exact same code.

```
import javafx.scene.chart.LineChart;
import javafx.scene.chart.XYChart;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
...
@FXML
private LineChart<Number, Number> ecgChart;
private XYChart.Series<Number, Number> seriesEcg;
...
seriesEcg = new XYChart.Series<>();
seriesEcg.setName("ECG");
for (int i = 0; i < 10; i++)
   seriesEcg.getData().add(new XYChart.Data<>(i, 0));

comboBox1.valueProperty().addListener(new ChangeListener<String>() {
   @Override
    public void changed(ObservableValue<? extends String> observable,
        String oldValue, String newValue) {
      ...
      if (newValue != null){
         switch (newValue){
            ...
            case "ECG":
                  if (!ecgChart.getData().contains(seriesEcg)){
                     ecgChart.getData().add(seriesEcg);
                     ...
                    }
                ...
            }
        }
    }
});

Timeline updateGraphs = new Timeline(new KeyFrame(Duration.millis(100),
    (ActionEvent event) -> {
   try {
      if (seriesEcg.getData().size() >= 10){
         seriesEcg.getData().remove(0);
         ...
      }
      Device.Frame data = frames[0];

      seriesEcg.getData().add(new XYChart.Data<>(signalsCounter,
          transEcg(data.an_in[ecgPos]))); //transfer function
      ...
      signalsCounter++;

   } catch (Exception ex) {
      logger.log(Level.SEVERE, "Error while updating charts", ex);
   }
}));
updateGraphs.setCycleCount(Timeline.INDEFINITE);
updateGraphs.play();
```

Code snippet 3.3. Sensors data visualization

After the above steps the first prototype was ready and able to read physiological signals and camera video. On the other side, Unity could make the examiner follow the participant's visual experience. Figures 4.3 and 4.4 show what Prototype 1 looks like. An example scene was used for Unity.

3.1.3 Second prototype

The second prototype new functionality is the recording one. Once the examiner is able to see everything in real-time, he might want to record a session. In order to keep several over time, a new folder is created every time a new session is recorded. The folder contains camera video, physiological data and VR video, together with some text files containing synchronization information. Following the same order as with first prototype, implementation started with camera.

Camera

As well as visualization, JavaCV library also provides recording functionalities. The class that handles recording is *FFmpegFrameRecorder* which, as the name suggests, internally makes use of *FFmpeg*. This is a complete and powerful software tool for recording, converting and streaming audio and video (see section 3.2).

For what concerns the user interface, a button was added to allow the examiner to start and stop the recording. First we need to initialize a new recorder object specifying output path, video resolution and number of audio channels. After setting some other parameters, the recorder can be started. This happens when the user clicks the *Start recording* button. In order to actually record a frame, we need to call the *record(Frame frame)* method on the recorder passing as an argument the frame captured in prototype 1 (see code snippet 4.1). This method is called inside the *CameraRunnable* class created before. A *record* boolean was used to indicate when the recording is active. After being recorded, the files need to be synchronized. The synchronization algorithm is the one illustrated in section 2.3. The information about the initial and final timestamps is written in a text file inside the directory of the current session. When the recording starts, the current timestamp is acquired through the *System.currentTimeMillis()* function, which returns

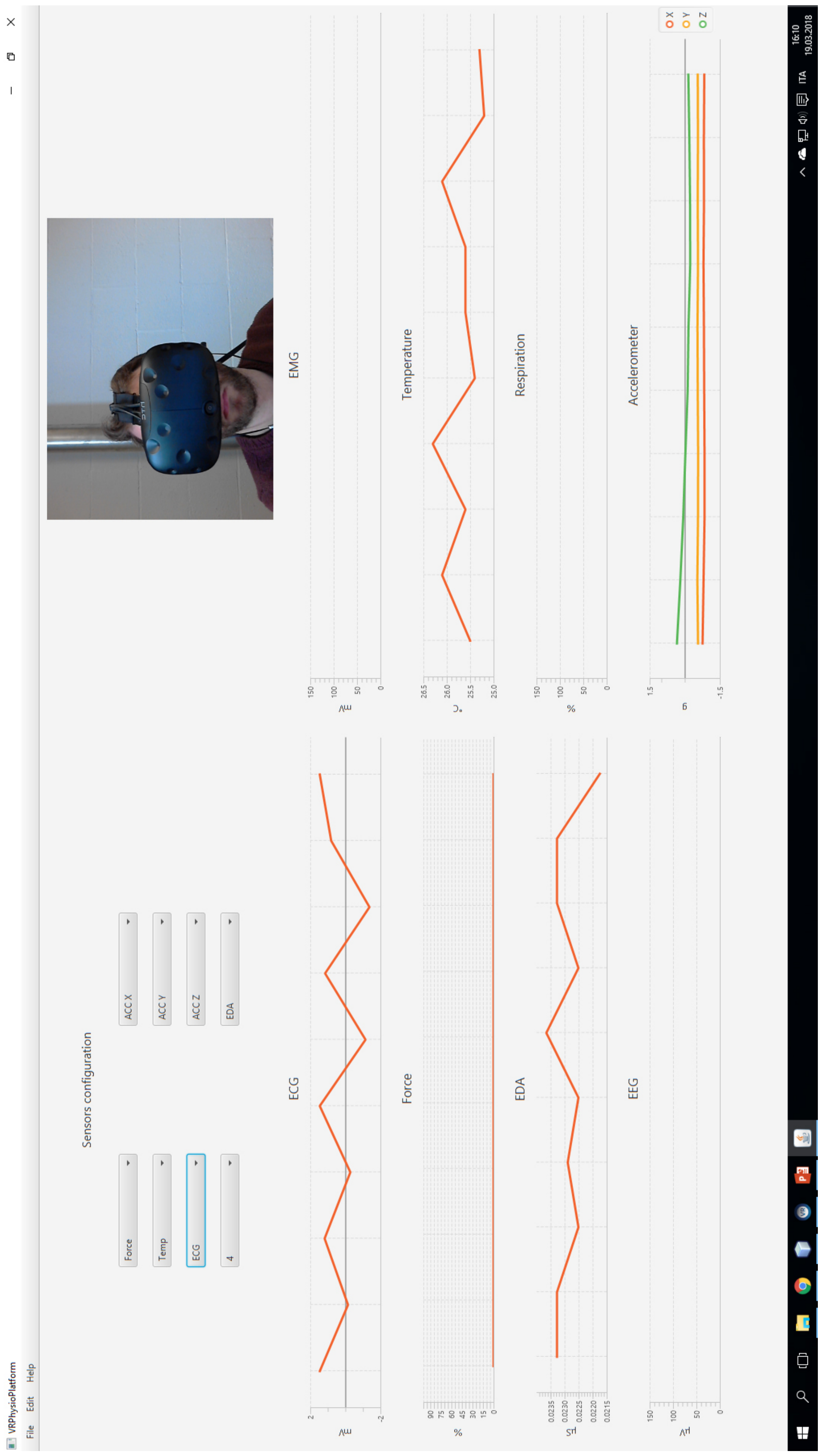
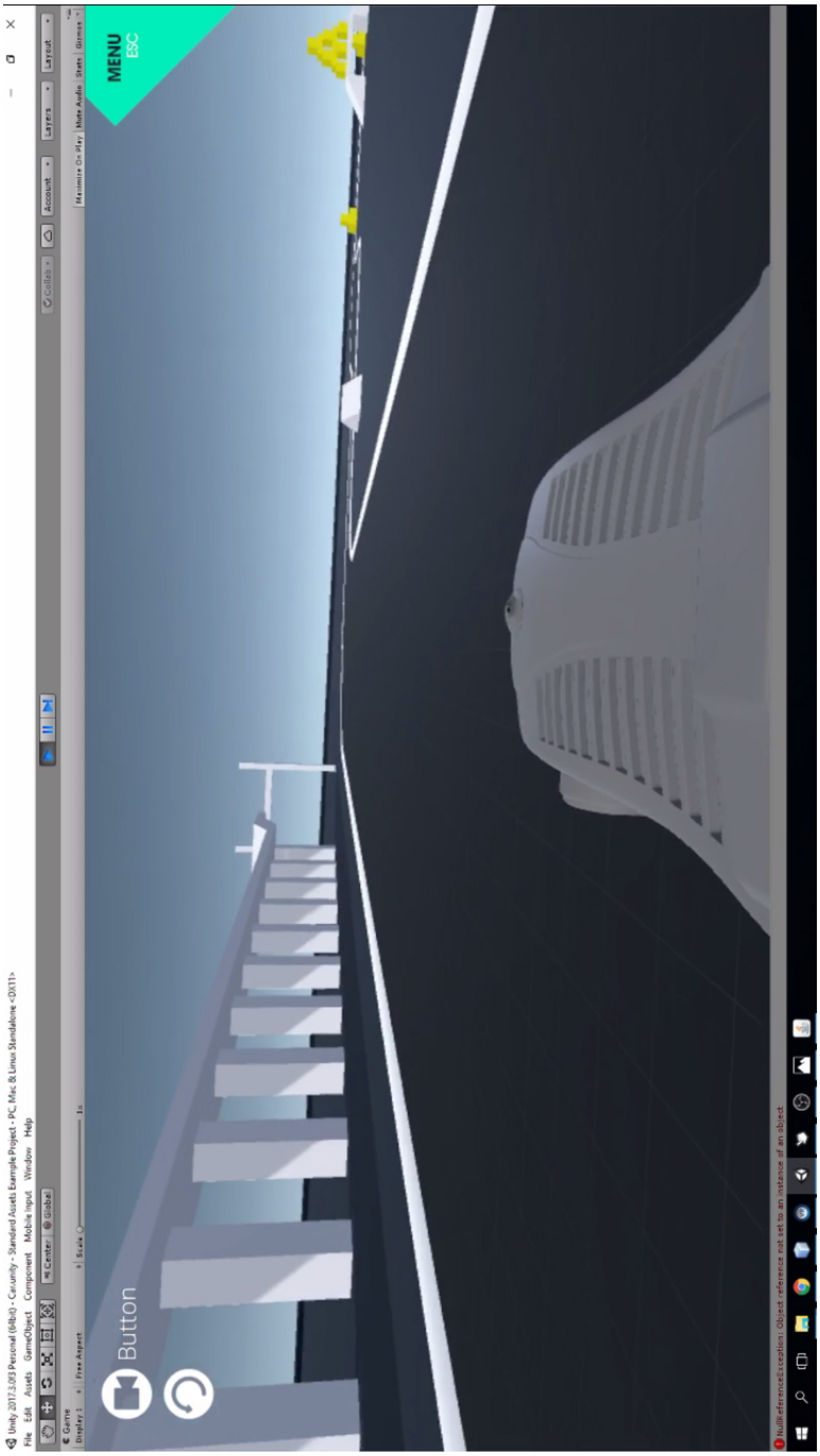


Figure 3.3. First prototype - Java application


Unity 2017.3.0f3 Personal (64bit) - Car.unity - Standard Assets Example Project - PC, Mac & Linux Standalone <DX11>
File Edit Assets GameObject Component Mobile Input Window Help
Center Global
Account Layers Layout
Game
Display 1 Free Aspect Scale 1x
Maximize On Play Mute Audio Stats Gizmos
MENU
ESC
Button

NullReferenceException: Object reference not set to an instance of an object

Figure 3.4. First prototype - Unity

the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC. As said before, the granularity of the value depends on the underlying operating system: since we are operating on Windows 10, the value will have a 10 milliseconds accuracy. This is not a big deal because a few dozen delay in synchronization is undetectable by humans and therefore is definitely acceptable. Instead of the final timestamp, in the text file was stored the duration in milliseconds of the recording. This metadata was saved as a JSON tree containing both camera and signals information. Camera fields were initial timestamp, duration (milliseconds), frame rate, size (kilobytes) and path of the file relative to the application main directory. Signals fields instead were initial timestamp, sampling rate, number of samples and path of the file relative to the application main directory. The following example shows the structure of the camera and signals metadata file.

```json
{
   "cameraVideo":{
      "timestamp":1526219413241,
      "frameRate":30,
      "duration":160367,
      "size":19465223,
      "path":"\20180513_154834\cameraVideo.mp4"
   },
   "signals":{
      "timestamp":1526219413241,
      "samplingRate":1000,
      "samples":158671,
      "path":"\20180513_154834\signals.txt"
   }
}
```

Code snippet 3.4. Recorded files metadata example

Code snippet 4.4 contains portions of code that implement what stated above. For details about synchronization see section 3.1.3.

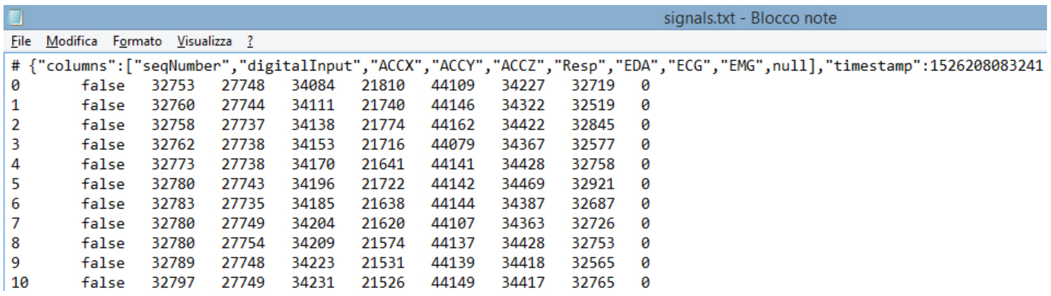
```java
import org.bytedeco.javacv.FFmpegFrameRecorder;
...
private Timestamp timestamp;
private FFmpegFrameRecorder recorder;
private boolean record = false;
private static long startTime = 0;
...
recorder = new FFmpegFrameRecorder(
               cameraVideoPathTemp,
               captureWidth, captureHeight, 0);
recorder.setVideoCodec(avcodec.AV_CODEC_ID_H264);
recorder.setFormat(MP4);
recorder.setFrameRate(FRAME_RATE);
... // Additional recorder settings
...
private void startRecording() {
   try {
      recorder.start();
      record = true;
   } catch (FrameRecorder.Exception ex) {
      logger.log(Level.SEVERE, "Error while starting camera recording", ex);
   }
}

private void stopRecording(){
   record = false;
   recorder.stop();
   ...
}
..
class CameraRunnable implements Runnable{
   @Override
   public void run() {
      ...
      if (record){
         startTime = System.currentTimeMillis();
         timestamp = new Timestamp(startTime);
         ...
         try {
            recorder.record(capturedFrame);
         } catch (FrameRecorder.Exception ex) {
            logger.log(Level.SEVERE, "Error while recording the frame", ex);
      }
    }
}
```

Code snippet 3.5. Camera recording

Physiological signals

For recording physiological signals we can just extend the reading functionality presented above. Saving sensors data in fact is not hard, since data frames can be easily represented as strings, thus they can be written in plain text files. The idea is that when we acquire the signals (once every millisecond), instead of just plotting the value (every 100 milliseconds) we can write it on a file. The same *record* flag as before can be used to control this behavior. If it is true then we can store the value, otherwise we just save it in a variable. Data frame is formatted as a tab-separated string containing sequence number, digital input flag and actual signals data. Before writing the frames, we insert in the file a line containing signals metadata such as timestamp, sampling rate and, most importantly, names of the used sensors. This allows to know which specific sensor each column refers to and is also the reason why we need combo boxes to configure the signals channels. Ax example signals file is the one in figure

Taking into account all these considerations, the signals recording code results as just an extension of the one reported in code snippet 4.2.



```
# {"columns":["seqNumber","digitalInput","ACCX","ACCY","ACCZ","Resp","EDA","ECG","EMG",null],"timestamp":1526208083241
0       false   32753   27748   34084   21810   44109   34227   32719   0
1       false   32760   27744   34111   21740   44146   34322   32519   0
2       false   32758   27737   34138   21774   44162   34422   32845   0
3       false   32762   27738   34153   21716   44079   34367   32577   0
4       false   32773   27738   34170   21641   44141   34428   32758   0
5       false   32780   27743   34196   21722   44142   34469   32921   0
6       false   32783   27735   34185   21638   44144   34387   32687   0
7       false   32780   27749   34204   21620   44107   34363   32726   0
8       false   32780   27754   34209   21574   44137   34428   32753   0
9       false   32789   27748   34223   21531   44139   34418   32565   0
10      false   32797   27749   34231   21526   44149   34417   32765   0
```

Figure 3.5. Saved signals file example

```java
import java.io.FileWriter;
import java.io.IOException;
...
private FileWriter signalsWriter;
...
if (!devs.isEmpty()){
   try {
      signalsWriter = new FileWriter(signalsPathTemp);
   } catch (IOException ex) {
      logger.log(Level.SEVERE, "Error while creating signals writer", ex);
   }

   int[] counter = new int[1];
   Thread getSignals = new Thread(() -> {
      try {
         dev.GetFrames(1, frames);
         if (record){
            Device.Frame frame = frames[0];
            String output = counter[0] + "\t" + frame.dig_in + "\t";
            int length = frame.an_in.length;
            for (int j = 0; j < length-1; j++){
               output += frame.an_in[j] + "\t";
            }
            output += frame.an_in[length-1];

            signalsWriter.write(output + System.lineSeparator());
            counter[0]++;
         }
      } catch (BPException ex) {
         logger.log(Level.SEVERE, "Error while getting the frames", ex);
      } catch (IOException ex) {
         logger.log(Level.SEVERE, "Error while writing signals data", ex);
      }
   });
}
```

Code snippet 3.6. Signals recording

Unity

For what concerns Unity VR recording, we can use the *RockVR* plugin (see section 3.2). It allows to record the visual output of the game or to send it to a streaming server. There are some customizable options such as output directory and frame rate, but to achieve the application's recording and synchronization goals there is the need of some adjustments. Basically we need the plugin to save the file inside the same session directory as the Java application and to write metadata such as initial timestamp and duration into a text file. The directory selection is an issue because we do not know if the examiner first starts the application or Unity. If he first starts the Java application, this has to create a new folder and Unity has to recognize and use it; if he first starts Unity, the opposite should happen. The solution was to use a "status" text file, placed in framework root directory, that contains the information about which application created the directory. In this way when the other one reads the file it understands that it must not create a new directory and it must use the already existing one.

The recording functionality is obtained editing parts of some RockVR scripts in order to make the desired behavior possible. The scripts are written in C# but, since they basically do the same things, the logic of the code is almost the same as the Java application one, so it is not explicitly reported in this section. Recording starts when the user presses the R key on the keyboard, timestamp is obtained using *DateTime.UtcNow* and the metadata JSON file has the exact same structure as the "camera" one in code 4.4.

The general execution flow is the following: Unity starts and checks the "status" file, creating the directory if needed, then waits for the examiner to start the recording. When it happens, timestamp is saved in the text file and recording begins; at the end of the session duration and size are added to the file.

Synchronization

Once the recording is stopped from both Unity and the Java application, the created files need to be synchronized. As explained in the design chapter, synchronization is based on system timestamps and files are cut according to timestamps difference. Here are presented some implementation details about it.

First of all, synchronization cannot be executed automatically by the framework at the end of the recording. The reason is that files take some time before being permanently and correctly saved on the hard drive, depending on the session length. This means that the synchronization phase cannot start immediately or after a fixed amount of time. The solution is to give the examiner the responsibility to start it by pressing a button: he has to check if the files are correctly saved and then he can start the process.

Synchronization is realized through the command line software*FFmpeg* (see section 3.2). There exist some Java wrappers of it but their usage was problematic, so the program was used directly from command line using the *Process* Java class. This allows to create a system process, launch it and wait for its completion. An FFmpeg distribution was downloaded and placed in the same directory as the Java application (see section 3.3) so that it is always available.

For what concerns the actual synchronization, first the system reads the meta-data files about camera and VR videos (camera and signals have the same timestamp), then it compares their initial timestamps and durations. Combining these two values we obtain four possible cases:

1. camera video starts first and lasts longer than VR video;
2. camera video starts first and lasts less than VR video;
3. VR video starts first and lasts longer than camera video;
4. VR video starts first and lasts less than camera video.

Let us analyze the first case. If camera video starts before VR video we need to cut its first portion in order to make the two videos start at the same time. If camera file lasts longer than VR file we also need to cut its final portion: by doing that we obtain two videos that start and end at the same time, thus they are synchronized. The same cuts we make on camera can be applied to signals file, since they share the same timestamp and duration. This reasoning can also be applied to case 3 simply reversing camera and VR video. Case 2 (and therefore 4) is a little bit more complicated because we need to cut the first part of the camera video and the last part of VR video, since it lasts more. FFmpeg is executed on a separate thread so that it does not interfere with the main application behavior. Code snippet 4.7 gives an overview of the different cases.

```
long cameraVideoTs = cameraVideo.get(TIMESTAMP).getAsLong();
long vrVideoTs = vrVideo.get(TIMESTAMP).getAsLong();

double vrVideoDuration = vrVideo.get(DURATION).getAsDouble();
double cameraVideoDuration = cameraVideo.get(DURATION).getAsDouble();

double diff;

if (vrVideoTs > cameraVideoTs){
   // camera video starts first
   diff = vrVideoTs - cameraVideoTs;

   double start = diff/1000; //seconds

   double cameraNewDuration = cameraVideoDuration - diff;
   if (cameraNewDuration > vrVideoDuration){
      // 1. camera video starts first and lasts longer than VR video

      double end = vrVideoDuration/1000;
      try {
         Thread cameraVideoCut = new Thread(new CutVideo(CAMERA_VIDEO,
             cameraVideoPathTemp, cameraVideoPath, start, end));
         cameraVideoCut.start();
        ...
        cutSignals(start*1000, end*1000);
        ...
```

```
          } catch (Exception ex) {
            logger.log(Level.SEVERE, "Error while cutting camera video", ex);
        }
    }
    else {
        // 2. camera video starts first and lasts less than VR video
    }
}
else {
    // VR video starts first
    diff = cameraVideoTs - vrVideoTs;

    double start = diff/1000;

    double vrNewDuration = vrVideoDuration - diff;
    if (vrNewDuration > cameraVideoDuration){
        // 3. VR video starts first and lasts longer than camera video
        ...
    }
    else {
        // 4. VR video starts first and lasts less than camera video
        ...
    }
}
```

Code snippet 3.7. Cut video cases

Cutting the signals file is very easy. Since we acquired data with a 1 kHz rate we have a line for each frame and frame counter represents time (e.g. 12500 refers to 12.5 seconds). This means that, assuming the difference between the timestamps is *diff*, we only need to read the file line by line, ignore the first *diff* lines and copy the others on a new text file. Same thing can be done to the last part of the file.

For what concerns the actual video cut task, a Java process launching *FFmpeg* was invoked. The process is created through the *ProcessBuilder* class, passing as parameters the string arguments that make up the complete command: path of the command to execute, input and output files path, starting and ending points. The class also allows to redirect the process output to the NetBeans console, letting the developer debug the application.

The problem with video cut is that it takes time. The first attempt with *FFmpeg* included the option *-codec copy*. This is very fast because makes the program skip decoding and re-encoding of the stream, but it is not accurate with respect to seeking. Video files are in fact composed of a number of *keyframes*, special frames that contain a whole image of the stream. Intermediate frames, those that are between two keyframes, only store changes from the previous keyframe and not the whole image. This means that we cannot cut a video between two keyframes without first decoding it, because the information is not complete. The first attempt tried to cut the video indicating a specific (up to milliseconds) start time and including the *-codec copy* option. This resulted in a video cut from the previous keyframe, with a significant error. Videos can in fact contain one keyframe for each 4 or 5 seconds of the stream. The only possible solution to this issue was to remove the copy option and decode the entire video, thus considerably slowing down the whole synchronization phase. This obviously depends on the length of the recording session but was considered not excessively long, so the solution was actually adopted. A progress bar was added to the GUI to inform the user about the completion of the synchronization job. Code snippet 4.8 shows the creation and the launch of the *FFmpeg* process.

```
private String ffmpeg;
...
ffmpeg = workingDirectory + "\\ffmpeg\\bin\\ffmpeg.exe";
...
class CutVideo implements Runnable{
   private final String video;
   private final String videoPathIn;
   private final String videoPathOut;
   private final double start;
   private final double end;
   ...
   @Override
   public void run() {
      ...
      try {
         Process process = new ProcessBuilder(
            ffmpeg,
            "-i",
            videoPathIn,
            "-ss",
            String.valueOf(start),
            "-t",
            String.valueOf(end),
            "-y",
            videoPathOut)
            .redirectErrorStream(true).start();

         StringBuilder strBuild = new StringBuilder();
         BufferedReader processOutputReader = new BufferedReader(new
             InputStreamReader(process.getInputStream(),
             Charset.defaultCharset()));
         String line;
         while ((line = processOutputReader.readLine()) != null) {
            strBuild.append(line).append(System.lineSeparator());
         }
         process.waitFor();
         ...
      } catch (Exception ex) {
         logger.log(Level.SEVERE, "Error while cutting the video", ex);
      }
   }
}
```

Code snippet 3.8. Cut video process

3.1.4 Third prototype

The third prototype adds the replay functionality to the system and brings it to its final version. The first thing to do is to implement the playback feature for camera and VR videos, then the one for the signals. Finally, we can add markers insertion functionality. In order to replay a session we first need to switch to *Offline mode*, then to open a folder containing the files. Both these tasks are executable through the application menu bar.

Playback

Camera and VR video can be played using the *MediaPlayer* class that provides the basic media controls such as play, pause, stop and seek. It works inside a *MediaView* graphical object and it just needs the path of the file to open. We use a single slider to give the user a visual overview on the playback progress. If a point on the slider is clicked, media players seek the corresponding time on the videos, and signals charts do the same. These behaviors are realized through property listeners that bind slider to media players and signals. Code snippet 4.9 gives an example of what said above and refers to VR video; the code for camera video is exactly the same.

```java
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;
import javafx.scene.control.Slider;
...
private MediaPlayer vrMediaPlayer;
@FXML
private MediaView vrMediaView;
@FXML
private Slider slider;
...
private void openVrVideo(String path) {
   if (path != null) {
      File file = new File(path);
      Media media = new Media(file.toURI().toString());
      vrMediaPlayer = new MediaPlayer(media);
      vrMediaView.setMediaPlayer(vrMediaPlayer);

      vrMediaPlayer.totalDurationProperty().addListener(
         (obs, oldDuration, newDuration) ->
            slider.setMax(newDuration.toSeconds()));
      slider.valueChangingProperty().addListener(
         (obs, wasChanging, isChanging) -> {
            if (! isChanging) {
                  vrMediaPlayer.seek(Duration.seconds(slider.getValue()));
            }
      });
      ...
   }
}
```

Code snippet 3.9. Play video

For what concerns physiological signals, it was used the same chart class as the one used for the *Real-time mode*. In order to play data fast and make seek possible, the whole text file containing signals data is stored in a list variable. In the large majority of cases this load does not affect system responsiveness because signals file is not big. Obviously if the recording session becomes very long (e.g. two hours) there may be some problems. The playback of the signals itself is realized with the same logic as the real-time acquisition one: *Timeline* class with 10 readings per second. The code is very similar to that in code snippet 4.3 so it is not reported here.

Markers insertion

The last thing to do is to implement the functionality that allow to insert a marker in a point of a session to denote an important event that occurred.

The idea is to save this information in a new text file as a *(timestamp, name)* pair. In this case timestamps are not the same entity as before: they indicate the number of milliseconds elapsed since the beginning of the video (exluding pauses), i.e. the time instant in the session when the event happens. Markers information is stored in JSON format (see example below).

```
{
   "markers":[
      {
         "timestamp":97361,
         "name":"Event1"
      },
      {
         "timestamp":10906,
         "name":null
      }
   ]
}
```

Code snippet 3.10. Markers file example

When the user wants to add a marker he clicks a button, playback stops and the corresponding timestamp is saved. A dialog window appears asking the user to insert a name; if this field is left blank the event will have a *null* name. The *(timestamp, name)* pair is then written on the text file. The point on the GUI when the marker is added is denoted by a red rectangle; if the mouse pointer moves over it a tooltip with the name of the event appears, if the mouse clicks on it, slider value is set to that point value.

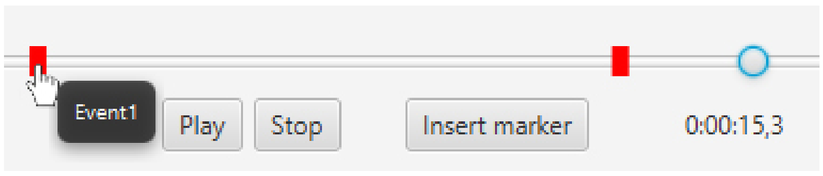


Figure 3.6. Markers appearance

Code snippet 4.10 shows the insertion of a marker on the GUI.

```java
import javafx.scene.control.TextInputDialog;
import javafx.scene.control.Tooltip;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.layout.AnchorPane;
import java.util.Optional;
...
@FXML
private AnchorPane anchorPane;
...
@FXML
private void insertMarker(ActionEvent event) throws IOException {
   pause();
   int timestamp = (int)(slider.getValue()*1000);
   Marker marker = new Marker(timestamp);
   TextInputDialog dialog = new TextInputDialog();
   dialog.setTitle("Event name");
   dialog.setHeaderText("Insert event marker");
   dialog.setContentText("Name of the event:");
   Optional<String> result = dialog.showAndWait();
   result.ifPresent(name -> marker.setName(name));

   drawMarker(marker);

   ... // add marker to markers list in the text file
}

private void drawMarker(Marker marker){
   Tooltip tooltip = new Tooltip(marker.getName());
   Tooltip.install(marker, tooltip);
   marker.setFill(Color.RED);

   marker.setOnMouseClicked((MouseEvent event) -> {
      double value = marker.getOffset()*slider.getMax()/slider.getWidth();
      slider.setValue(value);
   });
   ...
   anchorPane.getChildren().add(marker);
}
```

Code snippet 3.11. Insert marker

3.2 Third-party software

During the design and realization phase several third-party libraries and tools were cited. They represent a big resource for developers because allow to reuse software written by others, often already tested and improved. Despite most of them were already mentioned earlier, here they are described in details.

SteamVR plugin is a unity package that allows developers to integrate virtual reality into Unity projects. It is available for free on the official Unity Asset Store and works with both HTC Vive and Oculus Rift.

RockVR VR Capture is a plugin for recording video and audio of virtual reality Unity applications. It also supports multi-camera captures and 360 video captures. Like SteamVR plugin, it can be downloaded for free from the Unity Asset Store and works with both HTC Vive and Oculus Rift.

JavaCV is a utility that wraps commonly used libraries for computer vision such as OpenCV, FFmpeg, OpenKinect and others. Its source code is available on GitHub and it can be automatically downloaded with Maven.

Plux API is the official BiosignalsPlux API and allows to connect to the sensors and retrieve data. Besides Java, it is also available for C++, Python, C# and MATLAB, as well as Android and iOS. It can be obtained for free from the producer's website.

FFmpeg is one of the most famous and widespread frameworks for audio and video coding, decoding, filtering, streaming and playing. Its main component is a free and portable command line tool compatible with the majority of operating systems.

Gson is a Java library developed by Google for converting Java objects into their JSON representation. Its source code is available on GitHub and it can be integrated in a project using Maven.

JSON Object is a class representing the C# equivalent of Gson. Its code is available on GitHub but it can be directly downloaded from the Unity Asset Store.

Metadata extractor is a Java library for reading and extract metadata from image and video files. It supports a variety of standards such as Exif, XMP and IPTC but also formats like WAV, AVI, MP4, PNG, GIF. It can be downloaded with Maven.

Java MP4 Parser is a Java API for reading, writing and creating MP4 containers. It is able to mux audio and video, append recordings, change metadata and cut videos. The library can be retrieved through Maven.

3.3 Build and usage instructions

Once the development phase is completed, we need to build the application so that it can be deployed and used by other people. Since it is composed of two modules, the system will be distributed as a Windows installer that installs the Java application and a Unity package that can be imported and used inside a Unity project.

Java application

The application uses some Maven dependencies that we need to include in the output *jar* file. This can be done using the *Maven Shade Plugin*, which allows to create a jar package containing also all the dependencies. After this step, we need to be able to launch the package as a regular desktop application. The tool we can use for this purpose is *Launch4j*, a cross-platform tool for wrapping Java applications distributed as jars into Windows executable files (.exe). It also allows to add Java Virtual Machine (JVM) options, so that we can pass parameters to the execution of the program. We exploit this opportunity to specify to the JVM the path of the Plux dll file with the command *-Djava.library.path=./lib*. This folder needs to be distributed together with the executable file. Once the .exe file is ready, we need to create an installer that installs the program on the user's computer. This operation can be done using *NSIS* (Nullsoft Scriptable Install System), a tool used to create Windows installers from ZIP archives. Therefore we add the previously built executable file and the library folder to a new archive, together with the *ffmpeg* folder containing the software for video cut. We import the archive into NSIS and specify the default installation folder. The output of the tool is a self-extracting executable file that can install the framework very quickly and easily. The detailed steps to create the installer are the following:

1. build from NetBeans using the Maven Shade Plugin;
2. open the Jar file from step 1 into Launch4j and specify the output name;
3. inside the JRE tab, write 1.8.0 in the "min JRE version" field and a big number in the "max JRE version" field, since the app is not limited to run below a certain JRE version;
4. inside the JRE tab, write *-Djava.library.path=./lib* in the "JVM options" field;
5. build the wrapper;
6. create a ZIP archive with the executable file of step 5, the *lib* folder and the *ffmpeg* folder;
7. run NSIS and select "Installer based on ZIP file";
8. open the ZIP of step 6 into NSIS, specify an installer name, select "$PROGRAMFILES\YourNameHere" in the "Default folder" field and insert a name;
9. generate the installer file.

Unity package

The Unity package contains the SteamVR asset, the modified RockVR plugin, the JSON Object asset and some additional configuration files. Building the package is a trivial task because it can be done in the editor itself (select all the assets to include, right click on them and select "Export Package...").

When a user wants to use the package he has to import it and set some parameters in order to make it work properly. They basically consist in attaching video capture scripts to some game objects and adjust some of their properties. In the following list of required steps we assume that the right controller is used to control the recording but this can be changed without any constraints by the user.

1. import the package;
2. delete the *Main Camera* object;
3. place the *CameraRig* prefab (SteamVR/Prefabs) in the scene;
4. place the *CaptureCamera* prefab (RockVR/Vive/Demo/Prefabs) under *CameraRig/Camera (head)*;
5. attach *Properties* file to a fixed game object (e.g. *CaptureCamera*);
6. attach *CameraSampleCtrl* script (RockVR/Vive/Demo/Scripts) to *CameraRig/Controller (right)*;
7. attach *Vive_EventCtrl* script (RockVR/Vive/Scripts/Controls) to *CameraRig/-Controller (right)*;
8. attach *VideoCapture* script (RockVR/Video/Scripts) to *CameraRig/Camera (head)/CameraCapture*;
9. attach *AudioCapture* script (RockVR/Video/Scripts) to *CameraRig/Camera (head)/CameraCapture*;
10. in the *CameraRig/Controller (right)* inspector go to *Video Capture Ctrl*, set *Video Captures* size to 1, drag and drop *CaptureCamera* into *Element 0* and also into *Audio Capture*.

At this point Unity is ready to record sessions alongside the Java application.

Chapter 4

Validation

busvcubb

4.1 Proof of concept test

4.1.1 Development

4.1.2 Test protocol

1. Inform the participant about the test: it will be a VR scene, we will collect his physiological signals during the experience and we will record both the VR video and the participant's face with a webcam (3 minutes);

2. let the participant fill the consent form (2 minutes);

3. collect participant's data: age, sex, experience with VR, experience with videogames and with touchpad controllers (2 minutes);

4. describe the VR scene to the participant(30 seconds):

   > "You are in a dungeon of a castle and you have to find a key in the shortest possible time. The key is clearly visible once you reach the room where it is placed; you can just walk on it in order to get it";

5. describe the locomotion system to the participant (30 seconds):

   > "You can look around simply turning your head and you can walk using the controller touchpad. The reference for the locomotion is the headset direction, i.e. pressing on the positive Y axis will make you move forward with respect to the direction you are facing";

6. set up the sensors on the participant's body (5 minutes):

   - Accelerometer, placed on the chest through an elastic band;
   - Respiration, placed on the abdomen through an elastic band;
   - EDA, placed on the left hand index finger and middle finger;
   - ECG, placed on the left side of the chest, below the heart;

- EMG, placed on the neck, on the sternocleidomastoid muscle;

7. let the participant play an example VR scene (from VRTK example scenes) in order to get used to VR and locomotion system (3 minutes);

8. check if all the sensors are working properly through the official BiosignalsPlux application (OpenSignals);

9. prepare the test (1 minute):

   - check if the Java application task is still running from previous tests (if yes, kill it);
   - check if HTC Vive headset and controllers are ready;
   - check if physiological signals hub is turned on;
   - check if "status" file is correct;

10. run the system (1 minute):

    (a) run the Java application;
    (b) configure sensors inside the application;
    (c) check if signals are correctly acquired;
    (d) start recording from the application:
    (e) play the Unity scene;
    (f) start recording from Unity;

11. let the participant play the VR scene until he finds the key, then wait 30 seconds more (2 minutes);

12. stop recording (30 seconds):

    (a) stop recording from Unity;
    (b) stop recording from the Java application;
    (c) check if everything went fine;
    (d) stop the Unity application;

13. synchronize videos and signals (1 minute);

14. unplug sensors from the participant's body (30 seconds);

15. tell the participant not to tell anything about the experience to other people (30 seconds).

4.1.3 Results

4.2 Usability test

4.2.1 Protocol

Video tutorial shows how to run the Java application and the Unity scene, how to configure the sensors, how to start, stop and synchronize a recording session, how to switch to offline mode, how to open a previous recording, how to play it and how to add a marker to it.

1. Collect the participant's personal data: age, sex, education/professional training;
2. ask for expertise - on a scale of 1 to 5 (1 for no experience, 5 for expert) in the following topics:
    - data analysis;
    - physiological data recording;
    - virtual reality system use and setup;
    - programming.
3. let the participant watch the video tutorial;
4. participant can ask the examiner questions about things he did not understand from the tutorial;

    START PHASE 1
5.
6.
7.
8.
9.

4.2.2 Results

Bibliography

[1] B. K. Wiederhold, R. Gevirtz, and M. D. Wiederhold, "Fear of flying: A case report using virtual reality therapy with physiological monitoring," *CyberPsychology & Behavior*, vol. 1, no. 2, pp. 97–103, 1998.

[2] S. V. Cobb, S. Nichols, A. Ramsey, and J. R. Wilson, "Virtual reality-induced symptoms and effects (vrise)," *Presence: Teleoperators & Virtual Environments*, vol. 8, no. 2, pp. 169–186, 1999.

[3] S. Kuriakose and U. Lahiri, "Understanding the psycho-physiological implications of interaction with a virtual reality-based system in adolescents with autism: a feasibility study," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 23, no. 4, pp. 665–675, 2015.

[4] S. Seinfeld, I. Bergstrom, A. Pomes, J. Arroyo-Palacios, F. Vico, M. Slater, and M. V. Sanchez-Vives, "Influence of music on anxiety induced by fear of heights in virtual reality," *Frontiers in psychology*, vol. 6, p. 1969, 2016.

[5] J. Llobera, M. V. Sanchez-Vives, and M. Slater, "The relationship between virtual body ownership and temperature sensitivity," *Journal of the Royal Society Interface*, vol. 10, no. 85, p. 20130300, 2013.

[6] M. D. Wiederhold, K. Gao, and B. K. Wiederhold, "Clinical use of virtual reality distraction system to reduce anxiety and pain in dental procedures," *Cyberpsychology, Behavior, and Social Networking*, vol. 17, no. 6, pp. 359–365, 2014.

[7] I. E. Sutherland, "A head-mounted three dimensional display," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pp. 757–764, ACM, 1968.

[8] Kasey Panetta, "Top trends in the gartner hype cycle for emerging technologies, 2017." `https://www.gartner.com/smarterwithgartner/top-trends-in-the-gartner-hype-cycle-for-emerging-technologies-2017/`, 2017. [Online; accessed 26-June-2018].

[9] Gartner, "Hype cycle." `https://www.gartner.com/it-glossary/hype-cycle`, 2017. [Online; accessed 26-June-2018].

[10] M. Slater, "A note on presence terminology," *Presence connect*, vol. 3, no. 3, pp. 1–5, 2003.

[11] D. A. Bowman and R. P. McMahan, "Virtual reality: how much immersion is enough?," *Computer*, vol. 40, no. 7, 2007.

[12] M. V. Sanchez-Vives and M. Slater, "From presence to consciousness through virtual reality," *Nature Reviews Neuroscience*, vol. 6, no. 4, p. 332, 2005.

[13] J. S. Casanueva and E. H. Blake, "The effects of avatars on co-presence in a collaborative virtual environment," in *Annual Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT2001). Pretoria, South Africa*, 2001.

[14] J. Tromp, A. Bullock, A. Steed, A. Sadagic, M. Slater, and E. Frécon, "Small group behaviour experiments in the coven project," *IEEE Computer Graphics and Applications*, vol. 18, no. 6, pp. 53–63, 1998.

[15] M. Slater, A. Sadagic, M. Usoh, and R. Schroeder, "Small-group behavior in a virtual and real environment: A comparative study," *Presence: Teleoperators & Virtual Environments*, vol. 9, no. 1, pp. 37–51, 2000.

[16] R. S. Kennedy, N. E. Lane, K. S. Berbaum, and M. G. Lilienthal, "Simulator sickness questionnaire: An enhanced method for quantifying simulator sickness," *The international journal of aviation psychology*, vol. 3, no. 3, pp. 203–220, 1993.

[17] G. E. Riccio and T. A. Stoffregen, "An ecological theory of motion sickness and postural instability," *Ecological psychology*, vol. 3, no. 3, pp. 195–240, 1991.

[18] S. Selcon and R. Taylor, "Evaluation of the situational awareness rating technique(sart) as a tool for aircrew systems design," *AGARD, Situational Awareness in Aerospace Operations 8 p(SEE N 90-28972 23-53)*, 1990.

[19] S. G. Hart and L. E. Staveland, "Development of nasa-tlx (task load index): Results of empirical and theoretical research," in *Advances in psychology*, vol. 52, pp. 139–183, Elsevier, 1988.

[20] J. E. Muñoz, T. Paulino, H. Vasanth, and K. Baras, "Physiovr: A novel mobile virtual reality framework for physiological computing," in *e-Health Networking, Applications and Services (Healthcom), 2016 IEEE 18th International Conference on*, pp. 1–6, IEEE, 2016.

[21] R. Cassani, H. Banville, and T. H. Falk, "Mules: An open source eeg acquisition and streaming server for quick and simple prototyping and recording," in *Proceedings of the 20th International Conference on Intelligent User Interfaces Companion*, pp. 9–12, ACM, 2015.

[22] S. Rank and C. Lu, "Physsigtk: Enabling engagement experiments with physiological signals for game design," in *Affective Computing and Intelligent Interaction (ACII), 2015 International Conference on*, pp. 968–969, IEEE, 2015.

[23] A. Vourvopoulos, A. L. Faria, M. S. Cameirao, and S. B. i Badia, "Rehabnet: A distributed architecture for motor and cognitive neuro-rehabilitation," in

e-Health Networking, Applications & Services (Healthcom), 2013 IEEE 15th International Conference on, pp. 454–459, IEEE, 2013.

[24] D. Baldassini, V. Colombo, D. Spoladore, M. Sacco, and S. Arlati, "Customization of domestic environment and physical training supported by virtual reality and semantic technologies: A use-case," in *Research and Technologies for Society and Industry (RTSI), 2017 IEEE 3rd International Forum on*, pp. 1–6, IEEE, 2017.

[25] C. Cepisca, F. C. Adochiei, S. Potlog, C. K. Banica, and G. C. Seritan, "Platform for bio-monitoring of vital parameters in critical infrastructures operation," in *Electronics, Computers and Artificial Intelligence (ECAI), 2015 7th International Conference on*, pp. E–7, IEEE, 2015.

[26] C. McGregor, B. Bonnis, B. Stanfield, and M. Stanfield, "Integrating big data analytics, virtual reality, and araig to support resilience assessment and development in tactical training," in *Serious Games and Applications for Health (SeGAH), 2017 IEEE 5th International Conference on*, pp. 1–7, IEEE, 2017.

[27] E. Bekele, D. Bian, J. Peterman, S. Park, and N. Sarkar, "Design of a virtual reality system for affect analysis in facial expressions (vr-saafe); application to schizophrenia," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 25, no. 6, pp. 739–749, 2017.

[28] S. Otsuka, K. Kurosaki, and M. Ogawa, "Physiological measurements on a gaming virtual reality headset using photoplethysmography: A preliminary attempt at incorporating physiological measurement with gaming," in *Region 10 Conference, TENCON 2017-2017 IEEE*, pp. 1251–1256, IEEE, 2017.

[29] P. R. KB, P. Oza, and U. Lahiri, "Gaze-sensitive virtual reality based social communication platform for individuals with autism," *IEEE Transactions on Affective Computing*, 2017.