Quantifying the Sense of Presence in Virtual Reality Using Physiological Data

Eduard de Vidal i Flores

Advisors: Caglar Yildirim (MIT CSAIL)
Fox Harrell (MIT CSAIL)

Tutor: Antonio Chica (UPC)

A thesis submitted in partial fulfillment of the requirements for the:

Bachelor's Degree in Informatics Engineering

Bachelor's Degree in Mathematics

July, 2023

Abstract

English

Physiological measurement of player experience (PX) during gameplay has been of increasing interest within game research circles. Sense of presence, which refers to players' psychological feeling of being in a virtual environment, is seen as a major factor influencing PX. This thesis work explores how physiological measurements relate to sense of presence while playing a virtual reality (VR) roleplaying game. We find a significant negative correlation with skin temperature, which we interpret as having to do with thermal discomfort being associated with lower levels of sense of presence in VR. A commonly-used non-invasive wearable device for physiological measurement is the Empatica E4 wristband, which offers multiple physiological metrics including electrodermal activity, heart rate and skin temperature. That said, the E4's integration with popular game engines such as Unity proves difficult due to obstacles such as non-obvious critical bugs in the library and limited documentation applicability within the Unity context. We thus present *E4UnityIntegration-MIT*, an open-source Unity plugin designed to mitigate the challenges associated with integrating the E4 into Unity projects. The plugin exposes the E4's API for interfacing with Unity C# scripts, thereby enabling realtime data collection and monitoring, and provides the affordance of saving the data in an external file for data analysis purposes. The study here presented, which relied on *E4UnityIntegration-MIT*, also serves as a validation study for the plugin.

Keywords: player experience, PX, presence, virtual reality, VR, virtual reality gaming, VR gaming, Unity, affective gaming, Empatica E4, psychophysiological measures, electrodermal activity (EDA), heart rate variability (HRV), skin temperature

AMS: 62-07

Català

Les mesures fisiològiques de l'experiència de joc (PX) durant el joc han anat incrementant en popularitat en les comunitats de recerca sobre jocs. La sensació de presència, que es refereix a la impressió psicològica d'estar en un entorn virtual, és vista com un factor principal influent en el PX. Aquesta tesi explora com les mesures fisiològiques hi estan relacionades mentre es juga a un joc de rol de realitat virtual (VR). En trobem una correlació negativa significativa amb la temperatura de la pell, la qual interpretem com a relacionada amb que la manca de confort tèrmic estigui associada amb nivells menors sensació de presència. Un dispositiu *wearable* no invasiu comunament utilitzat per a mesures fisiológiques és la pulsera E4 d'Empatica, que ofereix múltiples mesures fisiològiques incloent l'activitat electrodèrmica, la freqüència cardíaca i la temperatura de la pell. Tanmateix, la integració de l'E4 amb motors de joc populars com Unity resulta difícil a causa d'obstacles com errors de programari crítics gens evidents a la llibreria i la limitada aplicabilitat de la documentació en el context de Unity. Així doncs presentem *E4UnityIntegration-MIT*, un *plugin* de Unity de codi obert dissenyat per a mitigar els reptes associats amb integrar l'E4 a projectes de Unity. El *plugin* exposa l'*API* de l'E4 permetent-ne la interacció amb *scripts* C# de Unity, habilitant així el recull i seguiment de dades en temps real, i proveeix la funcionalitat de guardar-les en un fitxer extern per permetre'n l'anàlisi. L'estudi presentat, que va dependre de *E4UnityIntegration-MIT*, alhora en serveix de validació.

Paraules clau: experiència de joc, PX, presència, realitat virtual, VR, joc de realitat virtual, joc VR, Unity, joc afectiu, Empatica E4, mesures psicofisiològiques, activitat electrodèrmica (EDA), variabilitat de freqüència cardíaca (HRV), temperatura de la pell

AMS: 62-07

Español

Las medidas fisiológicas de la experiencia de juego (PX) durante el juego han ido incrementando en popularidad en los círculos de investigación de juegos. La sensación de presencia, que se refiere a la impresión psicológica de estar en un entorno virtual, es vista como un factor principal influyente en el PX. Esta tesis explora como las medidas fisiológicas estan relacionadas con ella mientras se juega a un juego de rol de realidad virtual (VR). Hallamos una correlación negativa significativa con la temperatura de la piel, la cual interpretamos como relacionada con que el disconfort térmico esté asociado con niveles menores de sensación de presencia. Un dispositivo *wearable* comúnmente usado para medidas fisiológicas es la pulsera E4 de Empatica, que ofrece múltiples medidas fisiológicas incluyendo la actividad electrodérmica, la frecuencia cardíaca y la temperatura de la piel. Sin embargo, la integración del E4 con motores de juego populares como Unity resulta difícil a causa de obstáculos como errores de *software* críticos nada obvios en la librería y la limitada aplicabilidad de su documentación en el contexto de Unity. Así pues presentamos *E4UnityIntegration-MIT*, un *plugin* de Unity de código abierto diseñado para mitigar los retos asociados con integrar el E4 en proyectos de Unity. El *plugin* expone la *API* del E4 permitiendo su interacción con *scripts* de C# de Unity, habilitando así la recolección y el seguimiento de los datos en tiempo real, y provee la funcionalidad de guardarlos en un fichero externo para permitir su análisis. El estudio presentado, que dependió de *E4UnityIntegration-MIT*, sirve al mismo tiempo como su validación.

Palabras clave: experiencia de juego, PX, presencia, realidad virtual, VR, juego de realidad virtual, juego VR, Unity, juego afectivo, Empatica E4, medidas psicofisiológicas, actividad electrodérmica (EDA), variabilidad de la frecuencia cardíaca (HRV), temperatura de la piel

AMS: 62-07

Acknowledgements

Heartfelt thanks to:

- Caglar Yildirim, for his wisdom and guidance as advisor, as well as his tolerance for my blunders. But, I had to do it wrong before getting it right.
- Fox Harrell, for leading such an innovative and interdisciplinary team, and taking a chance in welcoming me on board. But for his leap of faith in me, I would certainly not have been able to do research in such an interesting field!
- Eyup, and Jack, and Emma, and everyone else in the lab, for their conversation on a variety of topics, be it the Boston area weather, ChatGPT, ethics, or old Turkish; which I at times made run for a bit too long. But it was all so interesting, I couldn't resist.
- Antonio Chica, for selflessly agreeing to be my tutor at UPC, and tolerating my infinitely long monthly emails. But, there was so much to say.
- Toni Pascual, Carles Sora and Miguel Ángel Barja, who were instrumental in helping establish contact with MIT scholars to get me admitted, despite surely feeling some bewilderment when I expressed interest in doing game research. But I had to remain loyal to my dreams.
- CFIS, and FME, and MIT, who are, each in their own way, formidable in their ambition, for providing me with unmatched opportunities to learn - long you live, and high you fly. But only if you ride the tide...
- My parents, and my sister, and the rest of my family, for their company even when an ocean's distance away (well, sometimes; some were around in the US too!), and who I'm afraid I consistently forgot to regularly communicate with. But no matter, I'll soon be back!
- My longtime friends back home, whose efforts to remain in contact throughout these months I greatly appreciate; and especially Laia, Ana, and Pau, hardware companions during my bachelor's, whom by serendipity I coincided with here at MIT for a merry while. But also to the newfound friends at MIT VISTA, and their fun events!
- To luck - the most important skill - without which I wouldn't be where I am. May I continue to enjoy its favour.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Assessing Player Experience (PX) is a matter of importance in game research and development. As players' enjoyment of a game is one of its most important characteristics, it is natural game researchers would seek to investigate what it is provoked by and how it is best measured, and game creators aim to maximize game enjoyment in their works.

Many factors have been proposed to be involved in PX. Some of the most prominent are flow (meaning an engaging balance between a player's ability and the game's challenge [30]), immersion, and sense of presence. Nevertheless, research on this topic is active. The extent to which these factors are influential to PX, and even sometimes the precise definitions and differences between some of them, are not yet fully known or universally agreed upon.

There are a range of approaches to measuring PX, from more subjective methods such as interviews or questionnaires to more objective techniques like in-game analytics and psychophysiological measurements. The latter offers advantages such as not needing to interrupt the experience, nor introducing memory bias (by not requiring of players to recall their experience after the fact) [29]. Moreover, the immediacy in the availability of the data also allows for the possibility of integrating the live readings as a realtime input into the game; this can be (and has been [34] [11]) exploited to craft psychophysiological feedback loops, whereby game parameters are altered according to the player's state.

As would be expected, proportional to the potential of this method are explorations on how to relate these measurements with PX or related constructs [30] [25] [26] [15] [5] [34]. It is along these lines that the present work makes a contribution: we examine what relationships may be found between a set of physiological measurements and the sense of presence in a virtual reality game.

Common physiological measurements taken include electrodermal activity (EDA),

heart rate (HR), interbeat interval (IBI), heart rate variability (HRV), blood volume pulse (BVP), temperature, electroencephalography (EEG), electromyography (EMG), respiratory rate (RR) and eye tracking [29]. These metrics are obtained through the use of a range of commercial or research devices.

One such commonly used device [34], [39], [7] is the Empatica E4. It is an promising tool for this kind of research because of several factors:

- It comes in a non-invasive wearable wristband form factor.
- It simultaneously records several of the aforementioned measures, namely BVP, EDA, IBI, skin temperature, and HR. (Additionally, while not a physiological sensor, it also includes a 3-axis accelerometer) [13]
- It provides multiple methods of using or interfacing with it, ranging from out-of-the-box working cloud solutions to a streaming service program for Windows that can forward its data streams over TCP, or APIs for integrating it into an Android or iOS application.

The availability of an Android API, the *E4link for Android* library, is particularly advantageous due to the plethora of devices which run on an Android or Android-based operating system, many of which support games. Indeed, even some virtual reality (VR) headsets such as Meta's Quest offerings are Android-based. Thus, we identify the existence of a broader interest in and need for integrating the E4 wristband into game engines that support Android-based game releases, such as the ever-popular Unity [41].

The study presented here, in which physiological measurements were taken while a VR game was played on a Meta Quest 2, was no different, and the E4 was assessed to be an ideal candidate for the task. However, it is not straightforward to use the *E4link* library in Unity. While Unity provides ways of interfacing with Android libraries, and *E4link* is small and simple enough, the integration proves difficult in practice. There exists a non-obvious critical bug in the library that must be worked around, some subtle issues regarding threading, and the documentation is non-exhaustive and of limited applicability in the context of Unity. Additionally, the library cannot work within the Unity player. This means the application must be deployed (and debugged) on an Android device every time, making the development workflow slow and tedious.

These experiences ultimately motivated the current thesis work, constituting the *E4link for Android* integration code developed as a Unity plugin named *E4UnityIntegration-MIT* and publishing it as open source [2], which we regard as the other contribution of this project. With the plugin, we intend on satisfying the previously stated desire for E4 integration into Unity with a ready-made solution, sparing others the development difficulties and effort.

The project's contributions are, then, twofold:

1. *E4UnityIntegration-MIT*

   An Android Unity plugin to support psychophysiological data collection and storage as well as physiological feedback game loops targeted at the game research and development communities. It comes with useful built-in default functionality such as automatically connecting to an E4 and writing recorded data to a file and has been designed to be simple, easy to install, and easy to use.

2. Exploratory study addressing:

   > How are players' physiological responses related to their sense of presence during a VR roleplaying game?

   Carried out using *E4UnityIntegration-MIT*, the study also doubles as on the field verification of the plugin's suitability for one of its intended purposes, psychophysiological data collection and storage.

We regard these to be technical and experimental contributions, respectively. The content of the document is organized around them.

Part I: Theoretical Framework

In chapter 2 necessary background to contextualize the rest of the work is provided, covering a range of topics relevant to the project.

Part II: Technical Work

As the Empatica E4's role was central in this project, a short chapter 3 provides all necessary information on it. Then, chapter 4 goes into extensive detail on *E4UnityIntegration-MIT*; the plugin is available for viewing in full in Appendix A.

Part III: Experimental Study

The study's methodology is presented in chapter 5. An analysis of the results follows in chapter 6.

Lastly, a final chapter 7 provides some closing remarks on the project overall.

Part I

Theoretical Framework

Chapter 2

Relevant Literature

Several important concepts and related literature must be introduced before the present work can be properly contextualized.

The aim of this project (chapter 5) was to quantify the sense of presence of players using physiological data. Work already exists which examines similar relationships, such as correlations between player experience and physiological data. The intent of such research has typically been to infer players' affective state through these physiological measurements. Under the understanding that player's emotions are a central factor in determining their enjoyment of a game experience, such psychophysiological inferences offer insight into what affects player experience, ostensibly with the goal of exploiting this knowledge in future to improve game experiences. We expand on the selection of topics we just introduced in the following sections.

2.1 Affect and Affective Computing

Player's emotions have already been identified as crucial to their enjoyment of game experiences [30]. Affect, a closely related construct, and Affective Computing, the multidisciplinary field devoted to its study in relation to computer sciences, therefore deserve further discussion.

2.1.1 Affect

Affect is a psychological construct strongly related to emotion [30]. In the literature, a commonly used conceptualization of affect is the circumplex affect model [36]. In contrast to a basic emotions model (by which there exist a number of distinct, independent, basic emotions such as sadness, anxiety, anger, elation, tension and

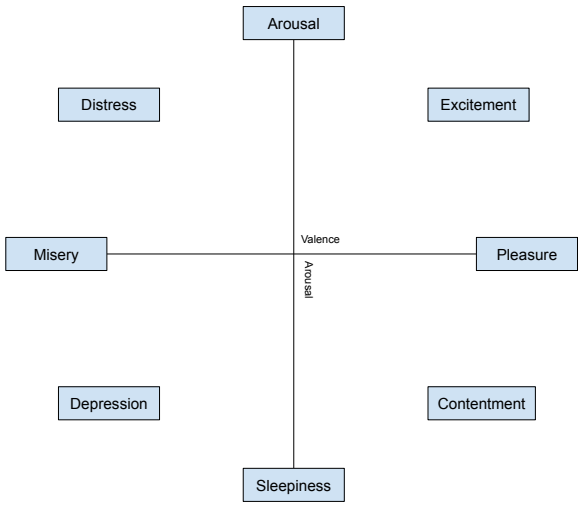

Figure 2.1: Circumplex affect model diagram

the like), the circumplex model places emotions roughly lying on a circle in a two-dimensional space whose orthogonal axes correspond to degree of valence (pleasure-displeasure) and degree of arousal (arousal-sleep), as in Figure 2.1. Thus, the emotions alluded to by common words like joy or depression can be situated, respectively, in high valence, high arousal, or low valence, low arousal, positions.

2.1.2 Affective Computing

Affective Computing (AC) is a multidisciplinary field encompassing computer science, engineering, psychology, neuroscience, and other disciplines. AC research investigates how affect influences interactions between humans and technology, informs understanding of human affect through affect sensing technology, and guides design, implementation and evaluations of systems involving affect [9].

It is clear, then, that investigations of the link between players' emotions and computer supported games, besides falling under game research, are also naturally tied to AC.

The quantification of player experience (through affect) by psychophysiological measurements (see subsubsection 2.2.2) is a technique of increasing interest in the game research communities, and is the approach explored in this project. Physiological measurements taken with this intent, often obtained through the use of research or commercial wearable devices, are therefore also closely linked to AC.

2.2 Player Experience

Player Experience (PX) is akin to user experience (UX) of products or services but specialized to games. Intrinsically related to players' emotions during a game experience, it is a concept of great interest in game research and development [30]. Several questions regarding PX are worth exploring, such as what may induce it or what other constructs are related to it, as well as how best to measure it.

2.2.1 Factors Affecting Player Experience

A range of other constructs have been identified as related to PX. While the extent of their relevance towards PX, or indeed the exact definitions and differences among them, are not necessarily fully understood or agreed upon, at least the following three bear mention: flow, immersion, and presence.

Flow

The state of flow [12] refers to the state of being felt when the balance between challenge and skill in an activity is struck; it is a pleasurable sensation of total involvement with a task. In relation to games, designing for it might revolve around developing an appropriate sequence of increasingly difficult conflict scenarios.

As it is a positive feeling, it is thought that achieving a state of flow during play is conducive to higher PX. As a matter of fact, the often used in game research Game Experience Questionnaire (GEQ) [33], "a self-report measure that aims to comprehensively and reliably characterise the multifaceted experience of playing digital games", includes a flow dimension whose purpose is to measure the degree to which this state of mind was present.

Immersion

Immersion refers to the intuitively understandable sense of being deeply involved in a game experience. There is no clear consensus regarding the exact definition of this concept, with several models having been put forward. Of these, [6] proposes a three-tiered division of immersion into three levels: engagement, engrossment, and total immersion. The last one, total immersion, seems to be identifiable with sense of presence, illustrating how it is sometimes difficult to precisely differentiate some of these concepts.

It is reasonable to postulate that experiences can be better enjoyed when one is

more involved with them, leading to the notion that in games immersion and PX may be related.

Presence

The sense of presence in relation to virtual environments refers to the feeling of actually being in the virtual environment. Factor analytic insights [37] suggest several notions are involved with the sense of presence, such as spatial presence (feeling present in the virtual space), involvement (attention paid to the virtual environment instead of the physical one), and realness (judgement of similarity between the real and virtual worlds).

Much like with immersion, in games the relation of sense of presence with PX, supported in for example [44], is credible since it is reasonable that experiences may be better enjoyed by those who feel more present in them.

2.2.2 Measurement Methods

There are a variety of methods to attempt to measure PX [29], from more subjective methods such as interviews or questionnaires to more objective techniques like in-game analytics and psychophysiological measurements. Each have their advantages and disadvantages.

Subjective Methods

Subjective methods are those for which human bias has a reasonable chance to enter the equation, such as interviews or questionnaires. Nevertheless, they oftentimes are the most accessible kind of measurement to administer, justifying their prevalent use [29].

Interviews are usually conducted by experts on the subject matter who know what information to probe for. While this has the upside of possibly revealing particularly relevant information, it has some disadvantages: participants may not be entirely honest with an interviewer, or, if there are several people being interviewed at once, such as in focus groups, be unwilling to give some information on account of others' presence, and, of course, the questions the interviewer chooses to ask influence the kinds of information that may be obtained to begin with.

As for questionnaires, among the most prevalent standardized ones is GEQ [33], which contains the subscales of Immersion, Tension, Competence, Flow, Negative Affect, Positive Affect, and Challenge. The degree to which the questionnaires are

able to robustly measure the factors they purport to is established by the empirical evidence gathered during validation studies.

Besides their relative lack of objectivity, subjective methods have further disadvantages [29]. One of them is introducing memory bias by requiring players to recall their experience after the fact. Another is that they must be employed either at the end of the gameplay (meaning there is little time resolution), or, conversely, if it is decided they are to be periodically carried out throughout the experience, they require repeated interruptions of the game.

All in all, though, subjective methods present several benefits not found elsewhere. They are sometimes the most reliable and validated ways of measuring PX or related constructs, and they are comparatively easy to carry out. Their usage is thus, unsurprisingly, widespread and commonplace.

Objective Methods

Objective methods encompass measurements such as in-game analytics and psychophysiological recordings. Due to their nature, they are much less susceptible to human bias than subjective methods.

In-game analytics can take many forms. Data such as, for example, how the player moved within the experience, where they pointed the game camera, what actions they took and when, and their performance in the game can be potentially useful for making inferences on the player's state during play.

Regarding psychophysiological recordings, these have been of increasing interest in game research circles in the past few years. Borrowing the approach from the field of AC (subsection 2.1.2), one may attempt to relate the measurements to players' emotional states during play to make inferences on PX; examples of this in the literature include [30] [25] [26] [15] [5] [34].

Common physiological measurements taken include: electrodermal activity (EDA), which measures the electrical conductivity of the skin; heart rate (HR); interbeat interval (IBI), which measures the time between heartbeats; heart rate variability (HRV), which measures the variations in HR; blood volume pulse (BVP), which measures the blood volume pulses resulting from heartbeats; temperature, as measured on the skin; electroencephalography (EEG), which measures the electrical activity of the brain; electromyography (EMG), which measures the electrical activity of muscles; respiratory rate (RR); and eye tracking [29]. A range of research and commercial devices are used to obtain these measurements.

In most cases the relationship between mental processes and body responses follow a so-called many-to-one relationship, meaning multiple psychological processes influence a given psychophysiological parameter. This situation is worse than a

one-to-one (one psychological process influences one psychophysiological parameter) relationship, but is nevertheless the most often used scenario in physiological evaluation [29].

Some of the common measurements warrant further review:

EEG EEG measures brain waves in different frequency bands such as alpha (associated with relaxation and lack of active cognitive processes, as well as information and visual processing), beta (related to alertness, attention, vigilance, and excitatory problem solving activities), theta (related to decreased alertness and lower information processing, though frontal midline theta activity in the anterior cingulate cortex scalp area also related to mental effort, attention, and stimulus processing), delta (most prominent during sleep, relaxation or fatigue), and gamma (still largely unexplored) [29].

EMG The most common kind of EMG in game research is facial EMG (fEMG). The brow muscke (corrugator supercilii) is used to indicate negative emotion and cheek muscle (zygomaticus major) to indicate positive emotion; for longer periods of play, the eye muscle (orbicularis oculi) is useful to detect high arousal pleasant emotions [29]. The choice of these muscles is due to their importance in facial expressiveness (that is to say, they are activated in certain patterns when smiling, frowning, etc.). These measurements are thus apt for determining affective valence (subsection 2.1.1).

EDA EDA is an exception in that it offers an almost one-to-one relation with physical arousal, from which we are able to ascertain the affective arousal (subsection 2.1.1) of a person. EDA measures changes in the passive electrical conductivity of the skin relating to increases or decreases in sweat gland activity, fluctuations which are caused by a person getting aroused by something they sense [29].

Changes due to the activity of the sweat glands are particularly interesting because they are controlled by the Autonomic Nervous System (ANS), a part of the Peripheral Nervous System (PNS) which is not under conscious control (as opposed to the Somatic Nervous System, which is and is responsible for, for example, willful activation of muscles). As such, this is a measure that is difficult to fake by participants, justifying why EDA may be regarded as an especially objective measure for arousal.

HR, IBI, HRV HR and IBI are obviously related measures (assuming HR is in beats per minute and IBI in seconds, at any moment $HR = \frac{60}{IBI}$, although it must be noted that HR is typically computed as the average accross a span of several seconds instead of instantaneously). IBI decrease, and consequently HR increase, is tied to increased information processing and emotional arousal [29]. HRV is a more complex measure to interpret, relating instead to changes in the frequency reflected by HR.

We thus see that there is good reason to believe in the endeavor of interpreting affective state, and thus inferring PX or others, from psychophysiological data, as there are ways of gleaning information about both affective valence and arousal from it.

While we have established that with respect to subjective measures they have the advantage of not introducing memory bias nor requiring that the gameplay be interrupted, they also have the additional benefit of being available in realtime. The theoretical implication of this is that the inference of PX or related concepts could be carried out live while playing a game, offering the opportunity to integrate this information into the game loop itself (presumably to dynamically adapt the experience so as to achieve an improved experience for the player), and, in fact, there already exist practical attempts at this. This notion is expanded upon in section 2.4.

2.3 Presence

Sense of presence has already been commented on (subsubsection 2.2.1). Central as it is to the present work, we delve slightly deeper into the topic.

2.3.1 Measurement Methods

Measuring sense of presence can be achieved through the use of questionnaires like the psychometrically validated one in [37]. The complete questionnaire consists of 13 items, all of which significantly load on sense of presence in general or on one of its identified factors (realness, spatial presence and involvement).

(A subscale of this questionnaire was used for the study (subsection 5.4.3), which was concerned only with sense of presence in general and the factor of spatial presence.)

2.3.2 Presence in Virtual Reality

It would be sensible that sense of presence would be amplified in VR experiences, as VR has been designed with immersiveness (whose relationship with presence is commented on in subsubsection 2.2.1) in mind. Indeed, studies like [10] find a significant improvement in sense of presence in two games from different genres (a racing game and a strategy game) in VR when compared to playing them on a desktop computer.

2.4 Psychophysiological Feedback in Games

Psychophysiological measurements, being available in realtime, allow games the enticing possibility of carrying out inferences regarding the player's state while playing a game, offering the opportunity of integrating this information into the game loop itself. Far from being an entertaining but remote theoretical prospect, practical attempts at this have been developed recently.

In the case of [34], which modified its game loop to take EDA and skin temperature into account, this was implemented with the intention of dynamically improving PX. While the result achieved was actually the inverse (PX was reduced when adapting the experience), the study nevertheless supports the idea that psychophysiological measurements can be used to influence PX (and, instead, further research should investigate how to create a positive, rather than negative, feedback loop for PX).

As for [11], EDA was integrated into the game loop of a VR game with the intention of improving user experience, and the results were increased desire to use, flow appropriateness, competence, and immersion with respect to the control group.

Part II

Technical Work

Chapter 3

Empatica E4

There exist a range of commercial and research devices capable of carrying out psychophysiological measurements. The Empatica E4 [20], Figure 3.1, was the device chosen for this project, and, considering much of the work revolves around it (and a variety of details concerning it), it is worthwhile to devote a chapter to it in order to give all necessary context.

The E4 is a noninvasive wearable wristband designed for research. Released in 2015, the E4 has since been used in a wealth of studies [1] concerning psychophysiological measurements.

3.1 Sensors

The E4 features an array of sensors that make it an attractive option as a noninvasive wearable device. The sensors include [20][14]:

Accelerometer 3-axis accelerometer.

Photoplethysmograph (PPG) Measures the amount of light emitted by an LED towards a tissue that reflects back to the sensor in order to infer blood volume changes.

EDA sensor Measures skin conductance.

Infrared thermopile Measures peripheral skin temperature.

These are demonstrated in Figure 3.2.

Figure 3.1: Empatica E4 [20]

Figure 3.2: Empatica E4 sensors [22]

3.2 Measurements

From the aforementioned sensors several measurements are produced, each having its own sample rate. The following are either automatically provided by the E4 itself or can be easily derived from those that are [14]:

3-axis accelerometer Data from 3-axis accelerometer sensor in the range $-2g\,\mathrm{m\,s^{-2}}$ to $2g\,\mathrm{m\,s^{-2}}$, sampled at 32 Hz.

BVP Data from photoplethysmograph (PPG), sampled at 64 Hz.

EDA Data from the electrodermal activity sensor in µS, sampled at 4 Hz.

IBI Inter beat intervals, as an intermittent output with $\frac{1}{64}$ s resolution.

Temperature Peripheral skin temperature (at the wrist on which the E4 is worn) in °C, sampled at 4 Hz.

HR Average heart rate values computed in spans of 10 s, sampled at 4 Hz. (This is not available directly from *E4UnityIntegration-MIT*, but can be computed from IBI.)

HRV Variations in heart rate. (This is not available directly from the E4 at all, but again can be computed from IBI.)

3.3 Interfacing Options

Empatica provides an assortment of ways in which to interface with the E4. They range from out-of-the-box solutions to utilities and APIs that allow integrating the E4 into custom applications.

The out-of-the-box solutions [20] are the *E4 manager* (a Windows and Mac application) and the *E4 realtime* (an Android and iOS application). These programs operate similarly, ultimately sending the recorded data to Empatica's secure cloud platform. Afterwards, the data is available for download by accessing *E4 connect* [16], the web portal to their cloud.

As for the alternatives that require programming [21], the *E4 streaming server* is a utility (available only for Windows) able to connect to multiple Empatica E4 devices and forward their data streams to clients connected over TCP [21], and the *E4link* library (for which two versions exist, *E4link for Android* and *E4link for iOS*) is an API that allows integrating the E4 directly into a custom (Android or iOS, as appropriate) application.

All options present benefits and drawbacks. While the obvious main advantage of the out-of-the-box solutions is that they are immediately available and functional, the fact that the data is uploaded to Empatica's cloud, which does not have API support for being accessed via code, means it is not as easily programmatically manipulated. Moreover, data is only available in the cloud after the entire recording is uploaded, and thus in this case there is no realtime data streaming potential. On the other hand, the other options do not suffer from these limitations, but require users to expend effort into developing their solutions.

The criterion by which we chose from among all the interfacing methods was to determine the simplest configuration that would satisfy our requirements (that is, the capability to receive the E4's data in realtime in an Android Unity application). Indeed, for this project a couple of the aforementioned interfacing options were used or at least considered. Those relevant to this work are further explained in their own sections below.

3.3.1 E4 streaming server

Recall from before that the *E4 streaming server* is a utility (available only for Windows) that can forward data streams from multiple E4 devices over TCP [21]; a diagram of this functionality can be seen in Figure 3.3. This way of interfacing with the E4 would have been sufficient to meet our requirements, but ultimately went unused in this project.

To use the program, first the E4 must be paired with it. Then, the data stream can be forwarded to one of the program's clients. Naturally, the client application must also have been designed with the capability of receiving the data stream in mind; however, this can be implemented without issue in an Android Unity application. In short, it is enough to fulfill our requirements.

Nevertheless, the development of the networking connectivity code in the client application is not the only cost of using this service. Indeed, its usage carries with it an increase in overall system complexity, since there must now exist an intermediate Windows computer running the server working as a bridge between the E4 and the client program. Moreover, there is also the added requirement of a Bluetooth LE adapter dongle for the Windows machine ("E4 streaming server works exclusively with the BlueGiga BLED112 Bluetooth Smart Dongle" [21]).

However, as the code to integrate the E4 was eventually to become a plugin of its own, we must take into account that dependencies on additional hardware render it less attractive. It is undesirable to restrict the ease with which game researchers and developers (intended users of the plugin) can ascertain whether the E4 can be satisfactorily integrated into their Unity project, as this may deter them from trying at all.

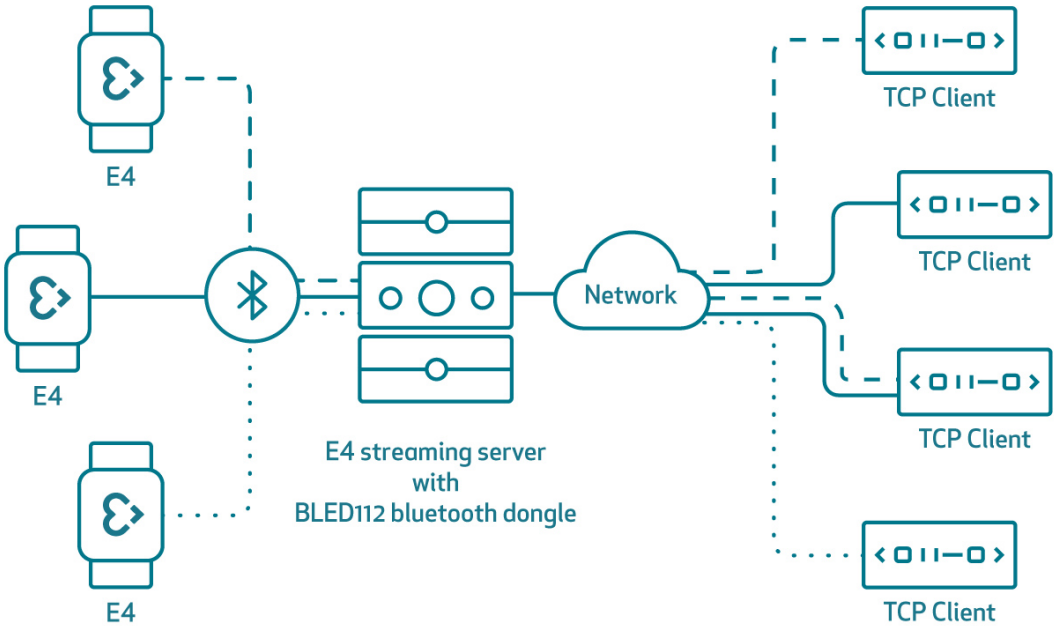

Figure 3.3: E4 streaming server use diagram [19]

To its credit, *E4 streaming server* offers features that are not available in other interfacing methods, in particular the ability to connect multiple E4 devices to an arbitrary amount of client devices. As we were not interested in such a capability, overall this utility does not offer the simplest solution and was thus disregarded in favor of other methods (subsection 3.3.3).

3.3.2 E4 realtime

E4 realtime is an application (Android and iOS) which can connect to an E4 device and start, stop, and upload its recording to Empatica's cloud; a diagram of this functionality can be seen in Figure 3.4. Using it is simple, as it only requires pairing the smartphone and E4 as setup, and it is controlled from a single button on the screen. Additionally, its user interface shows the measurements that the E4 is perceiving live, Figure 3.5, which is a great convenience for checking if the data collection is going correctly.

However, in spite of providing such a realtime feature *E4 realtime* actually fails to meet our requirements. Indeed, there is no way to communicate the data being received to an Android Unity application (in realtime or not; data is uploaded automatically to Empatica's cloud which, as mentioned before, exposes no API, so there is no programmatic way of transmitting it at all).

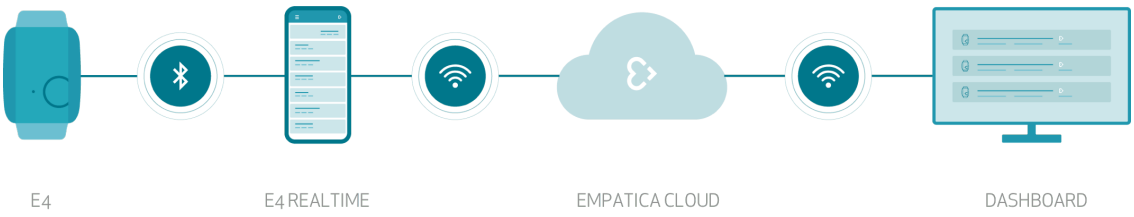

Figure 3.4: E4 realtime use diagram [20]

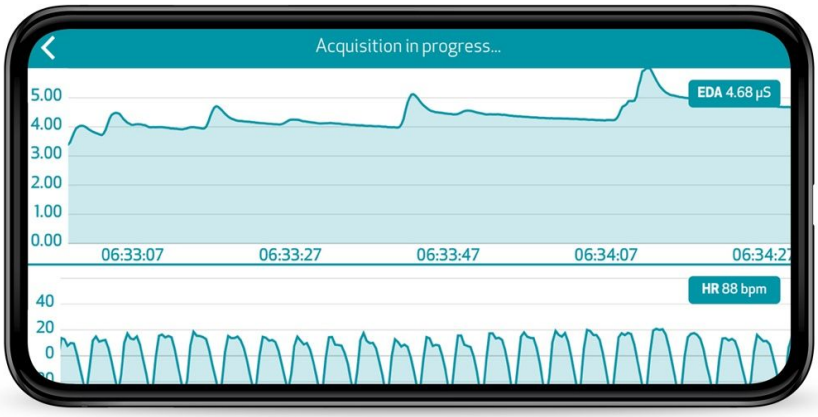

Figure 3.5: E4 realtime display [20]

Nevertheless, *E4 realtime* found its use in the present work. Its features made it a convenient tool for collecting baseline measurements (subsection 5.4.2) before the game measurements in the study (chapter 5).

Had the study been designed differently, however, and the baseline measurements taken inside the game (in a different scene preceding the main one, for instance), there would have been no reason to opt for using a different data gathering method for the baseline. In our case this was not done due to lack of time, despite the fact that it would have been beneficial at least in that it would have removed the need for a data format homogeneization step (subsection 5.4.2). This offers an advance into the projected usefulness of a single sufficient and simple method for meeting our needs, which is what *E4UnityIntegration-MIT* was made to achieve, and on which we elaborate on further elsewhere (chapter 4).

3.3.3 *E4link for Android*

E4link for Android is an Android library that provides an API to interface the E4 with a custom Android application. Its minimum Android SDK API Level is 19 (corresponding to Android 4.4 KitKat), and it works exclusively on 64 bit devices [21].

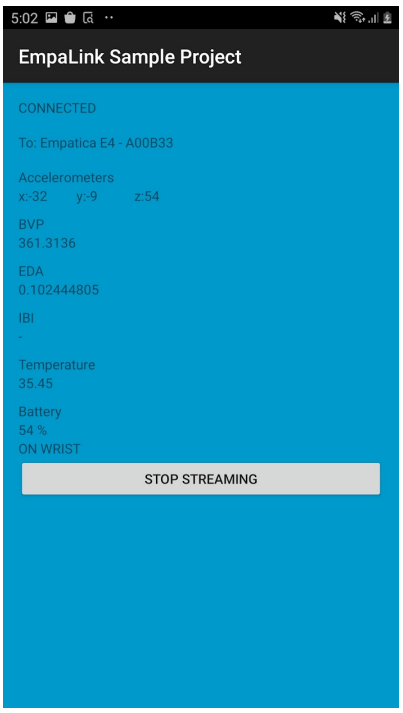

Figure 3.6: *E4link Sample Project* display [24]

This API presents itself as an ideal solution for our use case, as it can natively integrate the E4 directly with the Android Unity application, all while allowing receiving its data in realtime.

Compared to other solutions that were also sufficient (subsection 3.3.1) this one has the advantage of being the simplest possible. Indeed, it requires only an E4 and the device running the Android Unity program, both of which were already necessary pieces of hardware. Moreover, coding appears as though it ought to be simple as well, as this library is an officially supported API. (This ended up not exactly being the case (chapter 4), but was the perception at the time of choosing interfacing methods.)

As such, *E4link for Android* was the interfacing option of choice for supporting measurements during the game in the study (chapter 5), and the integration code is now constituted as the *E4UnityIntegration-MIT* plugin (chapter 4).

Unlike with E4 realtime, applications integrating *E4link for Android* will have a heterogeneous appearance as they are custom made. That said, Empatica do provide an Android *E4link Sample Project*, Figure 3.6, which can serve to showcase what the library is capable of.

Chapter 4

E4UnityIntegration-MIT

The majority of the technical work undertaken for this project is contained within *E4UnityIntegration-MIT*. It is an Android Unity plugin developed from scratch for integrating the Empatica E4 (chapter 3) into the Unity game engine. According to the reasoning in subsection 3.3.3, the plugin utilizes Empatica's *E4link for Android* API library to interface with the wristband.

Briefly (see more at section 4.4), *E4UnityIntegration-MIT* was designed to support the streaming of the E4's data through Bluetooth to Unity-made programs, with the intention of making the data available in realtime within the application. Foreseen potential uses for the data are inference of player emotional state to enable the live tailoring of the experience, or psychophysiological data collection and storage for later statistical examination.

In publicly releasing the plugin as open source [2] it is hoped that its intended users (the broader game research and game development community) will find in it a convenient tool for their purposes. At the same time, note that the plugin also satisfies this work's original need of a way to integrate the E4 into a Unity application running on the Meta Quest 2 (whose operating system is Android based). Thus, the second of the aforementioned potential uses is the one carried out in the study (chapter 5).

4.1 Integrating Android Libraries to Unity

Unity supports deploying to Android targets. It also allows the interfacing with Android libraries in several ways, with more having been added over the years, to enable using their functionalities within Unity projects. Any such Android library added to a Unity project is called an Android plugin.

The different types of Android plugins are [3]:

Android Library Projects The raw Android Studio project for a library, with a `.androidlib` extension.

Android ARchive plugins Libraries bundled in `.aar` files.

JAR plugins Libraries compiled to `.jar` files

Native plugins for Android They come in several types:

Plugins packaged in shared library files Possessing a `.so` extension

Plugins packaged in a static library Possessing a `.a` extension

Raw C/C++ source files Contained in `.c`, `.cc`, `.cpp` or `.h` files

Java and Kotlin source plugins Contained in `.java` or `.kotlin` files

All of these plugin types are integrated into Unity by placing their corresponding files into the `Assets/Plugins/Android` folder and marking Android to be their platform in the inspector's settings.

The *E4link* library is provided by Empatica bundled in a `.aar` (Android ARchive) format (section 4.2), and, as such, constitutes a supported library for Unity. However, as we shall also see, the Android ARchive format is not be the only plugin type in play in this project, justifying having given the previous overview.

In all cases save for the native plugins, making use of them from Unity C# scripts requires usage of the JNI (*Java Native Interface*) because we are calling Java code from outside of Java. Unity provides two APIs, one high level and one low level, to use JNI from C#. The former is sufficient for our purposes, and since it is simpler to use it is preferable to the latter. Thus, only the high level one is commented on.

Its high level API mainly provides three C# classes: `AndroidJavaObject`, which corresponds to a Java `Object`, `AndroidJavaClass`, which corresponds to a Java `Class`, and, `AndroidJavaProxy`, which allows for the implementation of Java interfaces by C# classes.

The first two classes allow performing the most usual interactions with objects and classes: getting and setting their fields, and calling their methods, all of them in static and non-static versions.

`AndroidJavaProxy`, on the other hand, is less straightforward. It is used by creating a C# class that inherits from it, and indicating in its constructor the Java interface that is being implemented. Then, the child class can implement the interfaces' methods. Crucially, this allows for the bidirectional communication between Java and C# code.

Triggering Java code execution from C# scripts is already possible through the use of `AndroidJavaObject` and `AndroidJavaClass`'s methods, but triggering C# code execution from Java scripts is only possible by implementing the following programming pattern, relying on `AndroidJavaProxy`:

1. In the Java code, define an `interface I`, a variable `I jv` of this type, and a setter function `void jvsetter(I i)` for this variable.
2. In the C# code, make a `class AjpI : AndroidJavaProxy`, which subclasses `AndroidJavaProxy`. Implement in it `interface I`'s methods. Call the previous setter and set it to an instance `AjpI csv` of the class, `jvsetter(csv)`.
3. In the Java code you can now call `jv`'s methods. The code executed will be the C# code implemented in `AjpI`'s definition.

Note that the last step means that, in effect, the Java code is triggering C# code execution.

The mechanism that has just been described is integral to the usage of the *E4link* library from within Unity. This is because, as elaborated upon in section 4.2, *E4link* provides two Java interfaces, namely `interface EmpaDataDelegate` and `interface EmpaStatusDelegate`, whose methods are called on connection status changes and data received events respectively. Thus, the task of integrating the library into Unity consists, fundamentally, on being able to run whatever C# code we want to execute upon having these methods invoked, which we achieve with the described pattern.

4.2 *E4link for Android*

The *E4link* library is not freely available for download. Instead, obtaining it requires visiting the Empatica connect web portal [16], where registered users can download it in the developer area [17] bundled in a *.aar* (Android ARchive) format.

Additionally, the API is not well documented. While Empatica's E4 link SDK for Android usage page [18] is available for the general public, it is not a complete reference but rather a streamlined tutorial. The provided JavaDoc documentation [23] is in a similar situation but is nonexhaustive as well. This is problematic, as even some of the methods belonging to critical interfaces in the library are not mentioned in any of these resources.

Instead, the author has found Empatica's sample Android application repository [24] source code to be the best official documentation available, as it showcases the library's use in an already functional program, together with decompiling the library

with Android Studio to analyse it. While they are good enough resources for those willing to dive deep and even do some reverse engineering (as, being realistic, may needed by programmers wanting to use the library in custom scenarios), there does not exist an adequate quick reference for those unfamiliar with it.

This is unfortunate, as in fact the API is relatively simple and easy to understand. Thus, it is reasonable to provide in this text an overview of the relevant parts of the library, with the goal of providing greater context for the subsequent sections.

Central to the library's usage are the following constructs:

- `class EmpaDeviceManager`, contained in the `com.empatica.empalink` package

  This is the main class of the package, allowing interacting with E4 devices. It supports a single connection with an E4. Some of its methods are:

  - `public EmpaDeviceManager(android.content.Context context, EmpaDataDelegate dataDelegate, EmpaStatusDelegate statusDelegate)`

    Class constructor.

    The `interface EmpaDataDelegate`, `interface EmpaStatusDelegate` implementing arguments determine the code to run on receiving data or status updates from the connected device. These interfaces are elaborated upon further below.

  - `public void authenticateWithAPIKey(java.lang.String key)`

    Each E4 device is associated to a list of developers (each with their own E4 connect account [16] and API key) that are allowed to use it. This is useful to protect E4 users' privacy (for example, it protects against a potential malicious program that automatically connects to any E4 it finds, regardless of whether the device belongs to the developer of the program or not, and which can therefore spy on the physiological status of whoever is wearing it). API keys are checked against Empatica's servers online; this is actually the reason for the plugin's requirement for internet connection in order to function.

  - `public void startScanning()`

    Starts scanning for E4 devices.

  - `public void stopScanning()`

    Complementary method for halting scanning.

  - `public void connectDevice(android.bluetooth.BluetoothDevice device) throws ConnectionNotAllowedException`

Connects to an E4; it throws `ConnectionNotAllowedException` if the connection to the device is not allowed.

- `public void disconnect()`

  Complementary method for disconnecting from the connected to device.

Typical usage of this class would be:

1. Instantiate it
2. Authenticate the API key
3. Wait to receive the `EmpaStatus.READY` status update in the status delegate
4. Start scanning for devices
5. Wait for callbacks in the status delegate regarding the discovery of a device
6. If this is the device one seeks to connect to, call the corresponding method to connect to it. (This should succeed as long as the API key authenticated with has permission to use the device.)

- `interface EmpaDataDelegate`, from the `com.empatica.empalink.delegate` package

  This interface must be implemented by some class in the *E4link for Android* client program's code.

  Its methods are all callbacks for when data fields from the E4 are received. Their arguments are the data value (or values) and the measurement's Unix time timestamp in seconds. Following is an overview of the relevant methods:

  - `void didReceiveAcceleration(int x, int y, int z, double timestamp)`
  - `void didReceiveBVP(float bvp, double timestamp)`
  - `void didReceiveGSR(float gsr, double timestamp)`
  - `void didReceiveIBI(float ibi, double timestamp)`
  - `void didReceiveTemperature(float t, double timestamp)`

  Note that there is no method to receive HR, even though this is one of the pieces of data provided when interfacing with the E4 by other methods, for example with *E4 realtime* (subsection 3.3.2). Still, this is not a problem as HR can always be deduced from IBI.

- `interface EmpaStatusDelegate`, from `com.empatica.empalink.delegate`

  This interface must also be implemented by some class in the *E4link for Android* client program's code.

Its methods are all callbacks for receiving status updates regarding devices. See below:

- `void didDiscoverDevice(android.bluetooth.BluetoothDevice device, java.lang.String deviceLabel, int rssi, boolean allowed)`

  Invoked when a new E4 device is discovered. The `boolean allowed` parameter is true if connection to this device is allowed according to the API configuration profile.

- `void didRequestEnableBluetooth()`

  Invoked when Bluetooth is detected as being off.

- `void didUpdateSensorStatus(EmpaSensorStatus status, EmpaSensorType type)`

  Invoked when the a device updates its status.

  Possible `enum EmpaSensorStatus` values are:

  * `EmpaSensorStatus.NOT_ON_WRIST`
  * `EmpaSensorStatus.ON_WRIST`
  * `EmpaSensorStatus.DEAD`

- `void didUpdateStatus(EmpaStatus status)`

  Invoked when the `EmpaDeviceManager` status changes.

  Possible `enum EmpaStatus` values are:

  * `EmpaStatus.INITIAL`
  * `EmpaStatus.READY`
  * `EmpaStatus.CONNECTED`
  * `EmpaStatus.DISCONNECTED`
  * `EmpaStatus.CONNECTING`
  * `EmpaStatus.DISCONNECTING`
  * `EmpaStatus.DISCOVERING`

- `void didEstablishConnection()`

  Not documented in the JavaDoc [23]. Presumably, invoked when connection to a device is established.

- `void didFailedScanning(int errorCode)`

  Not documented in the JavaDoc [23]. However, the *E4link Sample Project* code reveals that possible values for `int errorCode` are:

  * `ScanCallback.SCAN_FAILED_ALREADY_STARTED`

    For when a Bluetooth LE scan with the same settings is already underway by the app.

    - `ScanCallback.SCAN_FAILED_APPLICATION_REGISTRATION_FAILED`
      Fow when the app cannot be registered.
    - `ScanCallback.SCAN_FAILED_FEATURE_UNSUPPORTED`
      For when power optimized scan is not supported.
    - `ScanCallback.SCAN_FAILED_INTERNAL_ERROR`
      For when there is an internal error.

- `void bluetoothStateChanged()`

  Not documented in the JavaDoc [23]. Invoked when a Bluetooth adapter change is detected.

- `void didUpdateOnWristStatus(@EmpaSensorStatus final int status)`

  Not documented in the JavaDoc [23]. Presumably, invoked when the device starts or stops being worn on the user's wrist.

  This method, albeit undocumented, is actually unintendedly of great importance in this library, due to a nonobvious bug that involves it.

Much of the code related to basic usage of the library is actually dependent upon these methods. For example, a reasonable implementation would be to call `class EmpaDeviceManager`'s `public void startScanning()` method upon receiving `EmpaStatus.READY` as argument in

```
void didUpdateStatus(EmpaStatus status)
```

Likewise, `class EmpaDeviceManager`'s

```
public void connectDevice(android.bluetooth.BluetoothDevice
↪ device)
```

could be called upon receiving notification of a new device in

```
void didDiscoverDevice(android.bluetooth.BluetoothDevice
↪ device, java.lang.String deviceLabel, int rssi, boolean
↪ allowed)
```

In fact, this is the approach taken by the *E4link Sample Project*.

4.2.1 *E4link for Android* Dependencies and Requirements

Lastly, we remark on the fact that *E4link for Android* has a series of dependencies and requirements, all of which are inherited by projects using it. On the package dependency side, it relies on OkHttp [32] 2.7.5. As for requirements, it demands Bluetooth permissions, which themselves need Location access permissions, the

Bluetooth LE (Low Energy) feature, and Internet permissions. The implications of these necessities on *E4UnityIntegration-MIT*'s source files can be appreciated in subsection A.1.2, for the dependencies, and in subsection A.1.1, for the requirements.

4.3 An E4 Plugin: *E4UnityIntegration-MIT*

So far it has been explained that Unity supports integrating Android libraries into its projects, and that Empatica provide an Android API to interface with their E4 wristband. As such, E4 integration into Unity projects seems like it ought to be achievable in a simple manner. This begs the question: why, then, create *E4UnityIntegration-MIT*?

What was eventually to become the plugin's code was initially meant to be an ad hoc script for integrating the E4 into an already existing Unity project. Instead, it was the challenges along the way that forced it to be more complex and provided justification for considering it all its own standalone tool, as well as provided the motivation to release it publicly (that is, wanting to spare interested users the trouble).

Following is a list of difficulties encountered, some of which have already been hinted at. Hopefully by the end of the section it can be appreciated why there is benefit in releasing the code publicly as a solution that works right out of the gate. This way, users who wish to quickly integrate the device into their Unity program can do so without worry of encountering unforeseen difficulties, as was unfortunately the case for the present work.

4.3.1 Poor Documentation

It has already been commented on (section 4.2) that while there is some (incomplete) documentation [18] [23], to the author's taste the best references are the *E4link Sample Project* repository [24] together with decompiling the plugin's code in Android Studio.

This problem is only made substantively painful when in combination with the next issue.

4.3.2 Nonobvious Critical Bug involving a Method in *E4link*'s Interfaces

`void didUpdateOnWristStatus(@EmpaSensorStatus final int status)` is an undocumented method in `interface EmpaStatusDelegate`. (Note, though, that it is not a method in `interface EmpaDataDelegate`.)

However, at some point in `class EmpaDeviceManager`'s code the previous method is called from `interface EmpaDataDelegate`, which, needless to say, is incorrect behaviour and constitutes a bug[1].

This bug is critical because, when triggered, it causes the application to instantly crash. Moreover, we have anecdotally observed that this method seems to always be invoked immediately after creating an instance of `class EmpaDeviceManager`; if one instantiates this class at the beginning of the execution of the program (as was our case), this means the application crashes on startup.

Also very noteworthy is the fact that this bug forbids a pure C# implementation of *E4UnityIntegration-MIT*. Indeed, this bug can only be circumvented by ensuring that both interfaces are implemented by the same class (which is what is done in the *E4link Sample Project* [24]), making it so that the method is found despite being accessed through the incorrect one. However, Unity's `AndroidJavaProxy` C# class can only implement one interface, and therefore it is necessary to also have code outside of C# scripts (section 4.6).

Once again, this issue is further worsened by the next.

4.3.3 Testing and Debuggability Difficulty Inherent to Using *E4link for Android* within Unity

Typically, Unity applications, regardless of target platform, can be executed within the editor running on the development machine (and, indeed, this is the case with developing Android applications inside Unity too). This is useful both for quickly testing the application and for debugging it, as when running from the editor Unity supports a variety of features (breakpoints, stepping through the code in scripts, etc.) that result in a convenient debugging environment.

When adding *E4link* to the mix, though, these capabilities are lost. This is inevitable due to the library's reliance on specific hardware (Bluetooth LE) and soft-

[1]This bug was reported (on September 2020) and acknowledged by Empatica (on November 2020) [31] in the E4 link sample Android project repository. The issue was closed as completed on November 2020. Nevertheless, the latest version of E4 link available for download is from February 2020, before the report, and the bug is still present.

ware (Android environment) to perform its functions. These hardware and software features are expected from target devices, but cannot guaranteed to be available in development machines, so it makes sense that they are not supported from within the editor.

Consequently, it was found that when trying to integrate *E4link* the only way of testing the code was to deploy the entire project on an Android device each time. Building for and installing on an external Android device is a slow process, taking around 1 min to 2 min. Debugging is even more complex, requiring the use of the `adb logcat` [42] utility through the Android Studio IDE [4]. The tool reads every one of the many logged messages from the device, and one must attempt to make sense of the relevant ones (typically accomplished by filtering out the unrelated ones) to deduce where the issue lies.

It should now be made apparent why these three issues together make for an incredibly obtuse problem. As the application instantly crashes on startup, there are virtually no clues about what the problem is. It is therefore hard to figure out how to filter irrelevant `adb logcat` messages, and even more difficult to interpret those which are relevant, as they concern the calling of an undocumented method on an incorrect interface. Additionally, any changes made to the code to try to glean insight through trial and error incur once more the time cost of building and installing on an external device, significantly slowing debugging down.

4.3.4 Threading Difficulties

Unity's API (for general calls to the game engine) is only functional when used from the main Unity thread; using it from outside fails silently (that is, the calls do nothing). On Android, the main Unity thread is different from the main Android thread.

Usually, control flow within a Unity application is apparent to the developer, as all execution happens within the main Unity thread unless expressly programmed otherwise. However, *E4link* actually makes use of other threads for executing some of its code. To the best of the author's knowledge this fact is not documented anywhere; rather, this had to be discovered through reverse engineering (by decompiling the code, see section 4.2) instead.

Indeed, at certain points during execution *E4link* will post tasks to other threads. We found `interface EmpaDataDelegate` and `interface EmpaStatusDelegate`'s callbacks were, in practice, always running on threads other than the main Unity one, and as such all Unity API calls did nothing. This, unless handled properly, greatly limits how useful a callback's code can be.

The sensible way of dealing to deal with this behaviour is to employ a programming

pattern by which the callbacks only ever store in variables what the information they carried was, and then regularly poll these variables from the main Unity thread to react to their contents.

As such, while this difficulty can be easily sidestepped, it is nevertheless the case that for those unaware of this happening it can be very confusing. Considered in addition to the previous challenges, it must be said that being under the mistaken impression that no callbacks were ever being called (while in fact they were, but their code, consisting of Unity API calls which failed silently, did nothing) was an unwelcome additional hurdle.

4.4 Design Specification

While ad hoc solutions can get by with merely solving their original problem, making a tool that is convenient to use for a broader audience requires going somewhat further. Now that the reasons for having *E4UnityIntegration-MIT*'s code be published as a plugin have been stated, it is sensible to lay down explicitly what the design goals and intended use cases and audiences for the tool will be.

4.4.1 Functional Requirements

To support the streaming of the E4's data through Bluetooth in realtime on Android Unity projects.

This is basically synonymous with exposing `interface EmpaDataDelegate` and `interface EmpaStatusDelegate` to Unity C# scripts.

4.4.2 Foreseen Use Cases

1. Inference of the player's emotional state to enable live tailoring of the experience.

2. Psychophysiological data collection and storage for later statistical examination.

4.4.3 Target Audience

The target audiences of the plugin are the broader game research and development communities.

4.4.4 Nonfunctional Requirements

We want potential users to be able to quickly tell whether the E4 can actually be satisfactorily integrated into their Unity project. As such, we aim for:

Simplicity

The plugin consists of only a few files. Moreover, it has minimal dependencies, as only *E4link for Android* is needed.

Ease of installation

Installation is very simple: only a few project configuration changes are needed, and importing the plugin requires only dragging and dropping its files into the correct folder in the Unity project.

Useful default functionality

Ease of installation alone is not enough to allow users to get started quickly. Useful default functionality upon merely installing the plugin is paramount to letting users know that the tool is working and give them an accurate impression of whether it could fulfill their needs. As such, by default the program:

1. Asks for Bluetooth permissions if not already possessed.
2. Attempts to automatically connect to the first E4 it detects.
3. Records its data and stores it in a `.csv` file after a predetermined amount of time.

Note that this means the plugin automatically supports the second of its intended use cases (that is, "Psychophysiological data collection and storage for later statistical examination") right off the bat.

Ease of use

The code has been designed to be easily modifiable, and its default functionality serves as a contextual example regarding how to go about implementing desired behaviour.

4.5 *E4UnityIntegration-MIT* Overview

E4UnityIntegration-MIT consists of several files, Figure 4.1. Users can use the plugin by editing the source files directly.

It is simple to install *E4UnityIntegration-MIT* (for a full explanation see section A.2). All its files must be placed in the `Assets/Plugins/Android` directory of the Unity project. Then the following steps should be followed:

- *Enable Custom Main Manifest* in the project settings. If it is already enabled and a custom main manifest is already in use, then the plugin must be merged into the existing one, by copying over the lines related to permissions and feature requirements.
- *Enable Custom Main Gradle Template* in the project settings. If it is already enabled and a custom main gradle template is already in use, then the plugin must be merged into the existing one, by copying over the OkHttp [32] dependency (required by *E4link for Android*, subsection 4.2.1).

Lastly, needless to say Empatica's *E4link for Android* is also needed. It must be placed in the same folder, alongside the rest of the plugin's files.

We now provide an overview of each file's purpose and content.

4.5.1 `UnityE4.cs`

Unity C# script containing `public class UnityE4`. Attach the script to an empty GameObject to use the plugin.

Inside, two inner classes, named `class UnityE4linkDataDelegateCallback` and `class UnityE4linkStatusDelegateCallback`, expose the Empatica API. They implement the *E4UnityIntegration-MIT* interfaces defined in `UnityE4link.java` (subsection A.1.4) for receiving data and status updates, respectively. Users wishing to integrate the plugin to their use cases should alter the implementation of these classes' callback methods accordingly.

Note that, as noted in subsection 4.3.4, callback code may not execute Unity API calls, as they're limited to running on its main thread. Since callbacks are called by the E4 link library's code, which runs on a different thread, to react to received data or status updates users should instead store the information and trigger the desired behavior on the following Unity Update.

Additionally, this file also contains the code to record the received data periodically (default is every .25 seconds) and store it in a file.

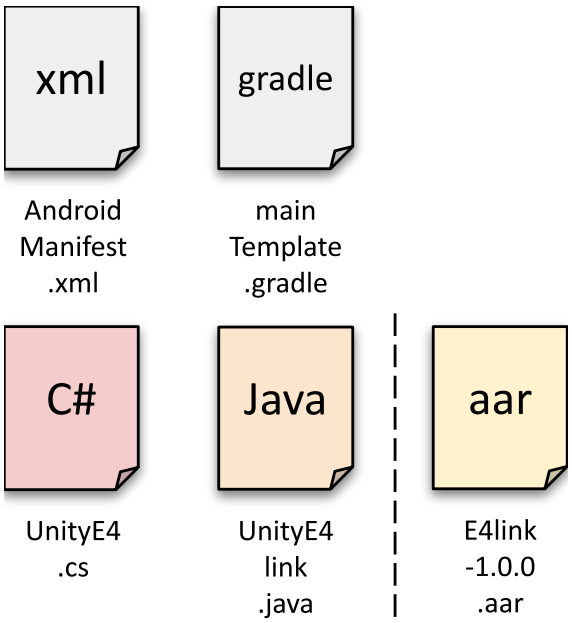

Figure 4.1: *E4UnityIntegration-MIT* files and dependencies. At the top are the build and configuration files that must be used or merged with existing ones. At the bottom, left of the dashed line, are the source files that may be edited by users to achieve desired functionality. To the right of the dashed line is Empatica's E4 link library; it is not distributed with the plugin and must be obtained directly from Empatica.

4.5.2 `UnityE4link.java`

Auxiliary Java file containing the UnityE4link class. It serves as a necessary bridge between E4 link and `UnityE4.cs`.

As explained in subsection 4.3.2, a critical bug forbids a pure C# implementation of the plugin. Its workaround is contained in this Java file (see subsection 4.6.1).

Being forced to introduce an intermediary Java file is not without advantages, however. While users may well get by without even looking at the Java file, the present implementation allows, for those unafraid to look, programming E4 link related functionalities directly in Java; at times this results in simpler code because often the alternative is having to call Java's utilities from C#. In fact, the provided default behaviour of connecting to the first E4 encountered is contained in this file.

Also, it was hinted at in section 4.1 that `.aar` (Android ARchive) format plugins (such as *E4link for Android*) were not the only type relevant to *E4UnityIntegration-MIT*. Indeed, by having this `.java` file we are in effect introducing a `.java` source plugin; this is the other type of plugin that was used for the tool.

4.5.3 `AndroidManifest.xml`

The plugin's custom main manifest.

Specifies the need for Bluetooth, internet, and location permissions (see subsection 4.2.1), as well as the Bluetooth low energy feature.

4.5.4 `mainTemplate.gradle`

The plugin's custom main gradle template.

Specifies *E4link for Android*'s OkHttp [32] dependency (see subsection 4.2.1).

4.6 Implementation

The difficulties described in section 4.3 and design objectives stated in section 4.4 shaped the plugin's code. This section is devoted to explaining in greater detail than in section 4.5 the characteristics and specific snippets of code that implement the workarounds and features of the plugin.

4.6.1 Working Around the Critical Bug in subsection 4.3.2

The workaround for the critical bug in subsection 4.3.2 is implemented in the Java file `UnityE4link.java`.

It is achieved by creating two interfaces, `interface UnityE4linkDataDelegate` and `interface UnityE4linkStatusDelegate`, which extend the delegates defined in *E4link for Android*, as well as inner `public class EmpaDelegate`, which implements both delegate interfaces from *E4link for Android* and defers its calls to objects of the previously stated interfaces. In this way, the necessary step of having both of *E4link for Android*'s delegates implemented by one same class is accomplished internally, without users who are only looking at the C# script needing to even know about it.

```
35      // Implement these interfaces in a Unity C# script to receive the E4's data
    ↪   and status updates
36      interface UnityE4linkDataDelegate extends EmpaDataDelegate { }
37      interface UnityE4linkStatusDelegate {
38          void didUpdateStatus(String status);
39          void didEstablishConnection();
```

```
40          void didUpdateSensorStatus(String status, String type);
41          void didDiscoverDevice(String deviceName, int rssi, boolean allowed);
42          void didFailedScanning(String error);
43          void didRequestEnableBluetooth();
44          void bluetoothStateChanged(boolean isBluetoothOn);
45          void didUpdateOnWristStatus(String status);
46      }
47
48      // Internal class, essential to work around a bug in the Empatica's E4 link
    ↪   library
49      // Passes callbacks through to the Unity C# script's
50      // Additionally, default behaviour of connecting to the first E4 found is
    ↪   implemented here
51      public class EmpaDelegate implements EmpaDataDelegate, EmpaStatusDelegate {
52          private final UnityE4linkDataDelegate unityE4linkDataDelegate;
53          private final UnityE4linkStatusDelegate unityE4linkStatusDelegate;
54
55          public EmpaDelegate(UnityE4linkDataDelegate unityE4linkDataDelegate,
    ↪   UnityE4linkStatusDelegate unityE4linkStatusDelegate) {
56              this.unityE4linkDataDelegate = unityE4linkDataDelegate;
57              this.unityE4linkStatusDelegate = unityE4linkStatusDelegate;
58          }
```

Listing 1: Fragment of `UnityE4link.java`: `interface UnityE4linkDataDelegate`, `interface UnityE4linkStatusDelegate`, and beginning of inner `class EmpaDelegate`

The rest of inner `class EmpaDelegate`'s code implements the *E4link for Android* interfaces' methods and passes them through to the newly defined interfaces; see subsection A.1.4 for the rest of inner `class EmpaDelegate`'s code.

4.6.2 Default Functionality

See subsubsection 4.4.4 for reference on what these features are.

Asking for Bluetooth permissions

Unless the user has already given an Android application Bluetooth permissions (for example, on a previous run) they need to be requested explicitly.

```
247     // Permissions section
248
249     private bool HasUnityE4linkPermissions()
```

```
250     {
251         return Permission.HasUserAuthorizedPermission(Permission.FineLocation);
252     }
253     // Call only if you lack the necessary permissions
254     private void RequestUnityE4linkPermissions()
255     {
256         PermissionCallbacks permissionCallbacks = new PermissionCallbacks();
257         permissionCallbacks.PermissionGranted += (string s) => {
    ↪ InitUnityE4link(); };
258         // FineLocation is critical, so always request again.
259         // However, not google's recommendation on how to handle user denying
260         permissionCallbacks.PermissionDenied += (string s) => {
    ↪ RequestUnityE4linkPermissions(); };
261         Permission.RequestUserPermission(Permission.FineLocation,
    ↪ permissionCallbacks);
262     }
263     // Call only if you have the necessary permissions
264     private void InitUnityE4link()
265     {
266         unityE4link = new
    ↪ AndroidJavaObject("edu.mit.virtuality.unityE4link.UnityE4link",
    ↪ unityE4linkDataDelegateCallback, unityE4linkStatusDelegateCallback,
    ↪ EMPATICA_API_KEY);
267     }
```

Listing 2: Fragment of `UnityE4.cs`: `class UnityE4`'s permissions section

```
270     void Start()
271     {
272         unityE4linkDataDelegateCallback = new
    ↪ UnityE4linkDataDelegateCallback(this);
273         unityE4linkStatusDelegateCallback = new
    ↪ UnityE4linkStatusDelegateCallback(this);
274 
275         // Check permissions and initialize unityE4link
276         if (HasUnityE4linkPermissions())
277         {
278             InitUnityE4link();
279         }
280         else
281         {
282             // When granted will initialize unityE4link
283             RequestUnityE4linkPermissions();
284         }
285     }
```

Listing 3: Fragment of `UnityE4.cs`: `class UnityE4`'s `void Start()`

The permissions section (Listing 2) contains support functions used within `class UnityE4`'s `void Start()` (Listing 3) method to ensure that Bluetooth permissions are possessed before instantiating and initializing `class UnityE4link`.

Automatically connecting to an E4

The code for automatically connecting to the first E4 discovered is a basic implementation of what is doubtless the most useful and important of *E4UnityIntegration-MIT*'s capabilities: connecting to an E4 device.

```
196     private final EmpaDelegate empaDelegate;
197     private final EmpaDeviceManager deviceManager;
198
199     public UnityE4link(UnityE4linkDataDelegate unityE4linkDataDelegate,
    ↪ UnityE4linkStatusDelegate unityE4linkStatusDelegate, String empaticaApiKey)
    ↪ {
200         Activity activity = UnityPlayer.currentActivity;
201
202         empaDelegate = new EmpaDelegate(unityE4linkDataDelegate,
    ↪ unityE4linkStatusDelegate);
203         deviceManager = new EmpaDeviceManager(activity.getApplicationContext(),
    ↪ empaDelegate, empaDelegate);
204
205         deviceManager.authenticateWithAPIKey(empaticaApiKey);
206     }
```

Listing 4: Fragment of `UnityE4link.java`: `class UnityE4link`'s constructor

```
105         // EmpaStatusDelegate methods
106         public void didUpdateStatus(EmpaStatus status) {
107             switch (status) {
108                 case INITIAL:
109                     break;
110                 case READY: {
111                     // Start scanning
112                     deviceManager.startScanning();
113                 } break;
114                 case CONNECTED:
115                     break;
116                 case DISCONNECTED:
```

```
117                     break;
118                 case CONNECTING:
119                     break;
120                 case DISCONNECTING:
121                     break;
122                 case DISCOVERING:
123                     break;
124             }
125 
126             unityE4linkStatusDelegate.didUpdateStatus(status.toString());
127         }
```

Listing 5: Fragment of `UnityE4link.java`: inner `class EmpaDelegate`'s `void didUpdateStatus(...)`

```
139         public void didDiscoverDevice(EmpaticaDevice bluetoothDevice, String
    ↪  deviceName, int rssi, boolean allowed) {
140             if (allowed) {
141                 // Stop scanning. The first allowed device will do.
142                 deviceManager.stopScanning();
143                 try {
144                     // Connect to the device
145                     deviceManager.connectDevice(bluetoothDevice);
146                 } catch (ConnectionNotAllowedException e) {
147                     // This should happen only if you try to connect when
    ↪  allowed == false.
148                 }
149             }
150 
151             unityE4linkStatusDelegate.didDiscoverDevice(deviceName, rssi,
    ↪  allowed);
152         }
```

Listing 6: Fragment of `UnityE4link.java`: inner `class EmpaDelegate`'s `void didDiscoverDevice(...)`

The code is written according to the guidance of Empatica's E4 link SDK for Android usage page [18] and follows in the footsteps of the *E4link Sample Project* [24]. It is virtually the same as the "reasonable implementation" described in item 4.2:

> For example, a reasonable implementation would be to call `class EmpaDeviceManager`'s `public void startScanning()` method upon receiving `EmpaStatus.READY` as argument in

`void didUpdateStatus(EmpaStatus status)`

Likewise, `class EmpaDeviceManager`'s

```
public void connectDevice(android.bluetooth.BluetoothDevice
↪ device)
```

could be called upon receiving notification of a new device in

```
void didDiscoverDevice(android.bluetooth.BluetoothDevice
↪ device, java.lang.String deviceLabel, int rssi, boolean
↪ allowed)
```

Data logging

The data logging functionality is designed to periodically (the default value chosen is 0.25 s) record the E4's measurements and store it in a `.csv` file after a predetermined amount of time (40.0 s). The fact that recording stops after this amount of time is arbitrary, and users of the plugin should modify the script to have the end condition correspond to the end of their measurements (which may be after a certain, possibly different, amount of time, or in response to an event such as the end of the experiment).

```
11      // Data logging section
12
13      private enum LoggerState
14      {
15          Initial,
16          Start,
17          Logging,
18          Stop,
19          DidWrite,
20          FailedWrite
21      }
22
23      private LoggerState loggerState = LoggerState.Initial;
24
25      // Tuple class is reference type so it is thread-safe to keep track of the
    ↪  last one
26      public Tuple<int, int, int, double> accLast = new Tuple<int, int, int,
    ↪  double>(0, 0, 0, 0.0d);
27      public Tuple<float, double> batteryLast = new Tuple<float, double>(0.0f,
    ↪  0.0d);
28      public Tuple<float, double> bvpLast = new Tuple<float, double>(0.0f, 0.0d);
29      public Tuple<float, double> gsrLast = new Tuple<float, double>(0.0f, 0.0d);
```

```
30      public Tuple<float, double> ibiLast = new Tuple<float, double>(0.0f, 0.0d);
31      public Tuple<float, double> tempLast = new Tuple<float, double>(0.0f, 0.0d);
32
33      private StringBuilder csvString;
34      private StreamWriter file;
35
36      private float currentSecond = 0.0f;
37
38      private IEnumerator E4DataLogger()
39      {
40          // Should now be loggerState == LoggerState.Start
41          GenerateCsvHeader();
42          loggerState = LoggerState.Logging;
43          while (loggerState == LoggerState.Logging)
44          {
45              LogConstruction();
46              yield return new WaitForSecondsRealtime(0.25f);
47              currentSecond += 0.25f;
48          }
49          // Should now be loggerState == LoggerState.Stop
50          if (WriteToFile())
51          {
52              loggerState = LoggerState.DidWrite;
53          }
54          else
55          {
56              loggerState = LoggerState.FailedWrite;
57          }
58      }
```

Listing 7: Fragment of `UnityE4.cs`: part of `class UnityE4`'s data logging section

```
126     private void UpdateLogger()
127     {
128         if (loggerState == LoggerState.Start)
129         {
130             StartCoroutine(E4DataLogger());
131         }
132     }
```

Listing 8: Fragment of `UnityE4.cs`: `class UnityE4`'s data logging section's `void UpdateLogger()`

```
287     void Update()
288     {
```

```
289         // Stop logging after 40 seconds
290         if (currentSecond >= 40.0f)
291         {
292             loggerState = LoggerState.Stop;
293         }
294         UpdateLogger();
295     }
```

Listing 9: Fragment of `UnityE4.cs`: `class UnityE4`'s `void Update()`

A state machine (whose states are Listing 7's `enum LoggerState`) controls whether logging is ongoing or not. Listing 7's `IEnumerator E4DataLogger()`, a Unity coroutine, is triggered (Listing 8's `void UpdateLogger()`) when logging is in its initial state, and runs every 0.25 s until it is stopped (Listing 9's `void Update()`) after 40.0 s.

4.6.3 Threading Considerations

C# callback code is executed on a thread different from Unity's main one, and thus, in addition to the typical care that must be taken when dealing with any multithreaded code, we must also ensure that no Unity API calls are made within callbacks (see subsection 4.3.4).

```
138     class UnityE4linkDataDelegateCallback : AndroidJavaProxy
139     {
140         private readonly UnityE4 unityE4;
141 
142         public UnityE4linkDataDelegateCallback(UnityE4 unityE4) :
    ↪ base("edu.mit.virtuality.unityE4link.UnityE4link$UnityE4linkDataDelegate")
143         {
144             this.unityE4 = unityE4;
145         }
146 
147         void didReceiveAcceleration(int x, int y, int z, double timestamp)
148         {
149             Tuple<int, int, int, double> accTuple = new Tuple<int, int, int,
    ↪ double>(x, y, z, timestamp);
150             unityE4.accLast = accTuple;
151         }
152 
153         void didReceiveBatteryLevel(float battery, double timestamp)
154         {
155             Tuple<float, double> batteryTuple = new Tuple<float,
    ↪ double>(battery, timestamp);
```

```
156             unityE4.batteryLast = batteryTuple;
157         }
158 
159         void didReceiveBVP(float bvp, double timestamp)
160         {
161             Tuple<float, double> bvpTuple = new Tuple<float, double>(bvp,
    ↪ timestamp);
162             unityE4.bvpLast = bvpTuple;
163         }
164 
165         void didReceiveGSR(float gsr, double timestamp)
166         {
167             Tuple<float, double> gsrTuple = new Tuple<float, double>(gsr,
    ↪ timestamp);
168             unityE4.gsrLast= gsrTuple;
169         }
170 
171         void didReceiveIBI(float ibi, double timestamp)
172         {
173             Tuple<float, double> ibiTuple = new Tuple<float, double>(ibi,
    ↪ timestamp);
174             unityE4.ibiLast = ibiTuple;
175         }
176 
177         void didReceiveTemperature(float temp, double timestamp)
178         {
179             Tuple<float, double> tempTuple = new Tuple<float, double>(temp,
    ↪ timestamp);
180             unityE4.tempLast = tempTuple;
181         }
182 
183         // Following method does not appear in E4 link documentation
184         void didReceiveTag(double timestamp)
185         {
186         }
187     }
```

Listing 10: Fragment of `UnityE4.cs`: Inner `class UnityE4linkDataDelegateCallback`

As we can see in Listing 10, inner `class UnityE4linkDataDelegateCallback` does nothing but store the received data into variables, and lets the program react to them on the main Unity thread (Listing 7's `IEnumerator E4DataLogger()`). This corresponds to the programming pattern prescribed in subsection 4.3.4.

Note additionally that in C# tuples are reference types [40], whose assignment is guaranteed by the standard to be atomic, and so it is thread-safe to keep track of the last one. That is to say, even if while writing to one of the tuples another thread

happens to be reading the values (in our case this would be the Unity main thread executing Listing 7's `IEnumerator E4DataLogger()` while one of the callbacks in Listing 10 is being invoked), it will either read from the old reference or the new one wholesale, but never some data fields from one and some from the other. This is important to preserve scientific data validity (if reads were mixed between the old and new measurements, we would in effect obtain an apparent measurement, created through combining data from both, which, in fact, never happened).

Part III

Experimental Work

Chapter 5

Method

The experimental contribution of the thesis involved investigating the psychophysiological correlates of the sense of presence in VR within the context of a roleplaying game. Specifically, the following research question was addressed:

> How are players' physiological responses related to their sense of presence during the roleplaying game?

For this purpose, the *E4UnityIntegration-MIT* plugin was integrated into a modified version of the On the Plane VR game [43] that was to be used in a study comparing the effects of perspective taking (by employing either an outgroup first person perspective or a third person perspective) on ingroup-outgroup bias in which psychophysiological data was also taken into account. The experimental study therefore additionally served as a validation of the plugin within the context of a real games reasearch study.

The present work's research question was in addition to the study's main research concerns. Therefore, while this section (chapter 5) on method describes the experiment in whole, including what all the various measured variables were, the following results section (chapter 6) analyses only the relevant fraction of all the collected data (namely, the E4 recordings and the presence scores).

5.1 On the Plane

On the Plane, Figure 5.1, is a roleplaying game that runs on desktop computers and the Meta Quest VR headsets [27].

It simulates an air travel experience in which two women from different backgrounds

Figure 5.1: On the Plane [43]. The left image corresponds to Sarah's first person perspective, whereas the right one corresponds to the third person perspective. The right one additionally depicts the menu that appears when giving players a choice between dialogue clauses.

are involved: Sarah, a muslim U.S. woman who wears a hijab, and Marianne, a woman from the U.S.'s Midwest who has had limited exposure to different cultures and customs [43, p. 208].

In the game, players are in control of Sarah's responses (through deciding between a limited set of dialogue clauses) while in conversation with Marianne. Players' choices affect the flow of the experience, as they cause changes in their standing in the simulation. Indeed, beyond just narrative differences, AI controlled characters are fully expressive and animated (they use body language, perform various gestures and can change their facial expressions), and thus can alter their behaviour according to the players' standing.

Preceding the previously described main part of the game is an orientation scene in which players are given instructions on the controls, as well as context regarding the simulated situation. For instance: "You are Sarah, a Muslim woman born in Indianapolis, IN. You travel frequently for work...", "You just visited family in Indianapolis and are on your returning flight to Newark, NJ..." [43, p. 208].

Participants were split into two condition groups for this study. While they played Sarah's role in both, in the first they played from Sarah's own perspective (first person), while in the second they played from the perspective of a flight attendant as a bystander (third person).

5.2 Design

The study used a between-subjects design with two conditions (Sarah's first person perspective or a third person perspective), whose differences can be also be seen in Figure 5.1. Several variables were recorded: physiological measurements such as EDA, HRV (derived from IBI), temperature, and EEG; and the subjective sense of

presence.

Other measurements, namely wrist accelerometer data and BVP, were taken too. However, these were collected only because the *E4UnityIntegration-MIT* plugin allowed doing so at virtually no extra cost. Indeed, it was expected that these variables would be uninteresting in relation to the research questions; however, ensuring all E4 data was measured was important to perform a complete verification of the plugin's capabilities.

Moreover, baseline measurements of physiological data (EDA, HRV, temperature, and EEG) were taken before participants played the game in order to account for differences between individuals.

5.3 Participants

Possible participants in this study were English-speaking adults located in the USA who had the ability to see visual content and navigate a virtual reality game. Participants were recruited in a variety of ways: advertisements in various mail and message groups, a post on the MIT Reddit forum [8], a call for participation in a class at MIT taught by one of the researchers, and in-person requests to participate. Thus, due to the nature of the explored communication avenues a large proportion of participants had some connection to MIT. Participants voluntarily registered by visiting an online sign-up page, where they could sign up for the study. Participants were compensated with a $10 Amazon gift card for partaking in the experiment.

A total of 41 participants (21 participants in Sarah's first person perspective, 20 in the third person perspective) took part in the study. Two participants' data were rendered unusable by technical glitches, resulting in a dataset of 39 participants (19 participants in Sarah's first person perspective, 20 participants in the third person perspective). Of these, 24 were female, 9 were male and 6 did not report. Participants' average age was 24.

5.4 Materials

A variety of materials were used to carry out the experiment. While the game employed has already been commented on (section 5.1), the following sections explore the rest of the tools utilized.

Figure 5.2: Meta Quest 2 [28]

5.4.1 Meta Quest 2

The Meta Quest 2 [28], Figure 5.2, was the VR headset used to run the game. As would be expected of a modern device, it has internet connection and Bluetooth capabilities, which enables it to connect with the Empatica E4.

5.4.2 Empatica E4

The Empatica E4, Figure 3.1, was used to capture several physiological measurements. Note that On the Plane is a Unity game [43], and the Meta Quest 2's operating system is Android based. Taken together, these two considerations mean this is an ideal situation in which to use *E4UnityIntegration-MIT*.

Indeed, the plugin was integrated into the game and used to capture the E4's data. As mentioned previously (section 5.2), the physiological measurements of interest were EDA, HRV (derived from IBI) and temperature; additionally, accelerometer data and BVP were gathered too for *E4UnityIntegration-MIT* verification purposes. All these fields were captured at a sampling rate of 4Hz.

As for the baseline, the E4 data was instead captured by the E4 realtime application running on a smartphone. *E4UnityIntegration-MIT* was not used for the baseline as these measurements happened outside of the game. As a consequence, data was not captured at a sampling rate of 4Hz but rather at the native sampling rate of the E4, which is higher than 4Hz for some data fields. To remedy this discrepancy, a Python script was developed that converted the data generated when using E4 realtime into *E4UnityIntegration-MIT*'s format in order to homogenize measurements.

5.4.3 Presence scale

In order to determine the players' level of presence a presence scale consisting of six items (taken from the thirteen items presented in [37]; only those relating to the constructs of Presence (paragraph 5.4.3) and Spatial Presence (paragraph 5.4.3) were used) rated on a 5-point Likert scale was employed. Ratings from each item were averaged to obtain a presence score for each participant.

Presence

- In the simulation world I had a sense of "being there"

Spatial Presence

- Somehow I felt that the virtual world surrounded me
- I felt like I was just perceiving pictures
- I did not feel present in the virtual space
- I had a sense of acting in the virtual space, rather than operating something from outside
- I felt present in the virtual space

The presence questionnaire was integrated into the In the Plane game. It automatically appeared after the participant had completed the main section of the game.

5.5 Procedure

Participants were invited to take a seat upon arrival. They were then briefed on the research study and presented with an informed consent form for signing. Participants could cease their participation at this time or at any other during the study. (One candidate did so.) Each participant was then assigned their condition randomly.

Baseline measurements followed. The Empatica E4 was setup, stressing that these are noninvasive sensors and do not hurt, and a roughly 1 minute video [35] was played on a tablet for them to watch while the measurements were taking place.

Experience with previous participants informed future procedure slightly in this case. In time, we learned to ask participants to silence their smartphones and smartwatches, and to not leave their devices on the table, as the vibrations against a hard surface would also make noise. In any case, these constituted minor distractions which should not have a noticeable effect on the measurements.

After baseline measurements came the On the Plane VR game. Participants were shown the relevant buttons on the controllers before helping them don the headset, which already had the game (the correct version of it, according to their condition) setup and running (waiting in the initial menu). The experience began when the researcher told the player their participant ID and they input it into the game. Interaction with the researcher would be kept to a minimum (only answering participants' questions) while they experienced the orientation and main scenes of the game. Measurements were taken during the main scene of the game.

Here experience informed procedure once again. Progressively more attention was paid to correctly adjusting the headset on participants, as this affects how clearly they see the virtual environment. In time it became clear that those unfamiliar with VR headsets would fail to realize they were not seeing clearly, or even that they were outside of the physical boundaries the headset establishes (the effect of this, which is supposed to be very apparent, is that the virtual environment loses color and becomes translucent, blending with the outside physical environment, in order to allow the user to determine where in the physical space they are and avoid getting hurt; one participant reported having this experience, and is thus excluded from further analysis on the grounds that immersion was compromised).

The presence questionnaire was completed right after the gameplay, as it automatically appeared after completing the main part of the game. Once participants were done, they were aided in taking off the headset and sensors and were presented with a post-experience demographic questionnaire on a computer.

Once done, the participant was thanked, given a $10 Amazon gift card as compensation, and dismissed. This process took participants approximately 20 minutes to complete.

Figure 5.3 and Figure 5.4 depict the procedure's setup from the point of view of a participant or an experimenter respectively; many elements described in this section can be appreciated in the images.

5.6 Research Ethics

Research involving human subjects falls under the purview of US federal regulations. In this case, this study is categorised as human subjects non-biomedical research.

Figure 5.3: Procedure setup - participant's view. Participants would sit in the chair closest to the camera, and would first read and sign the infromed consent form, then watch a video on the iPad during baseline measurements, and finally play the game on the Meta Quest 2.

Figure 5.4: Procedure setup - experimenter's view. The experimenter virtually always sat on the chair with the shirt hanging. Connected to the experimenter's laptop are the Empatica E4 and MyndPlay MyndBand, which are charging before the next participant arrives. To the left of the laptop are more printed copies of the informed consent form as well as a procedure instruction sheet for reference.

Thus, it was subject to review from an IRB (Institutional Review Board) at MIT for ethical and regulatory compliance, and all researchers interacting with participants were required to complete the appropriate "Social & Behavioral Research Investigators" course training.

Indeed, the call for participation advertisement text, informed consent form, and study procedure passed IRB approval. Accordingly, the rights of participants were observed when: candidate participants were informed and not mislead, during advertising or when receiving the informed consent form, regarding the purpose, procedures, risks and discomforts, and potential benefits of the research; participants voluntarily elected to partake, and could withdraw from the study at any time; and their privacy and confidentiality was protected, both pertaining to their collected data as well as their participation in the study in the first place, ensuring data was stored securely, and personal information and study data stored separately and associated only by an anonymous code.

Chapter 6

Results and Discussion

The guiding research question in this work was: "How are players' physiological responses related to their sense of presence during a VR roleplaying game?" The variables of EDA, HRV, temperature and presence score were analysed in order to ascertain any possible relationships between them. A variety of descriptive and inferential statistics tests were conducted to this end.

All physiological measurement variables were normalised with respect to their baseline (by subtracting the baseline mean from the experiment mean for each subject) in order to account for individual differences. Additionally, we note that condition (Sarah's first person perspective or third person perspective, see section 5.2), Table 6.1, showed no significant relationship with any of the considered variables in any of the tests. As such, the following sections report the data in bulk (sample size of 39), regardless of condition.

Condition	n
1st person	19
3rd person	20
Total	39

Table 6.1: Condition groups and sample sizes

Note that one of the variables analysed, HRV, was not directly measured (subsection 5.4.2). Rather, it was obtained from IBI by computing the HR, and from that the HRV, at any given time, and then considering its samples every 0.25s (thus mimicking the sampling method of *E4UnityIntegration-MIT*).

6.1 Descriptive Statistics and Pearson Correlations

Variable	M	SD	EDA	HRV	Temp.	Presence
EDA (µS)	.202	1.48	—			
HRV (beats/min)	1.50e-2	.309	-.13 p=.45 [-.42, .20]	—		
Temp. (°C)	.452	1.14	.09 p=.57 [-.23, .40]	.02 p=.90 [-.30, .33]	—	
Presence	3.88	.554	.22 p=.17 [-.10, .50]	.14 p=.38 [-.18, .44]	-.39 p=.02 [-.62, -.08]	—

Table 6.2: Means (M), standard deviations (SD), and Pearson correlations (coefficients, p-values, and 95% confidence intervals); n=39.

Correlation coefficients between physiological measurement variables (all rows except for the last in Table 6.2) are all small and have p-values above .05, supporting no linear relationship between them. This can be considered positive, as it means that the variables that will next be contrasted with presence do not appear to be dependent or correlated among themselves.

Presence, however, does have a significant correlation with temperature. Indeed, having a Pearson coefficient of -.39 (p=0.02), participants with a higher skin temperature had a significantly lesser sense of presence.

In this case, it is difficult to relate temperature variability as having much to do with the game itself (such as individual differences in the reaction to the game experience). Indeed, while it could be argued that participants more sensitive to Marianne's biased behaviour could have had a more heated reaction to the game, it would also be expected that those more affected would be those who report a higher sense of presence in the experience, rather than lesser.

Instead, we surmise the correlation may have to do with thermal discomfort. Our temperature variable refers to skin temperature on the wrist (at the position the E4 was at), and while it is not the same as core body temperature, it can be reflective of degree of thermal comfort [38]. As such, the likelier explanation is that some participants felt hot during the experience, affecting their sense of presence.

This leads us to formulate some practical suggestions on room temperature relating to presence. For starters, users wishing to maximise their sense of presence while playing games may want to pay attention to regulating the temperature of their environment (although if they are investing time and effort into playing for pleasure, ostensibly they may also already be interested in having an adequate room temperature to begin with for basic comfort reasons, if this option is available to them). Also, researchers wishing to measure sense of presence in a particular game (or, more generally, virtual environment) may want to also take into account temperature as part of the analysis as it has been shown to be relevant.

No other variable was significantly correlated with presence. While HRV is far from being significant (p=.38), EDA comes relatively closer (p=.17), with a Pearson coefficient of .22.

6.2 Other Variables

EDA, HRV, temperature and presence were not the only variables measured. While we are restricting our analysis to E4 data and presence scores, even then values such as acceleration magnitude (from the E4 accelerometer) and BVP (also from the E4) have not been reported on, because they are ultimately uninteresting to the research question at hand.

Nevertheless, it is worth briefly commenting on whether any of the variables are correlated. While average BVP is far from achieving significance in correlation (p>.05) with any other variable, the average magnitude of acceleration does correlate significantly (p=0.01) with HRV, with a coefficient of -.41. In other words, those who move around (their arm) more have a lower HRV, meaning a steadier or declining HR. Moreover, average magnitude of acceleration also correlates significantly (p=0.02) with IBI, with a coefficient of -.36. This means those who performed more movement had overall lower IBI (which is to say, higher HR).

Taken together, the interpretation of this could be that those who were more physically active during the experience had an overall higher, and steadier, HR throughout, and maybe even declining during play as they became tired (which could explain low average IBI and small or negative HRV). Alternatively, it could be a reflection of participants' nervousness. Recalling that the physiological measurements are normalised by subtracting their average baseline values, it could be that participants' agitation is reflected in a low or negative difference in IBI (participant is more nervous, meaning higher average HR, when playing the game than during the baseline), low or negative HRV (participants become increasingly relaxed as time goes by during the experience, as they become used to it, resulting in declining HR, and consequently in low or negative HRV), and increased average movement (of the

arm). We further note that a sizeable amount of participants reported no or little prior experience with VR. Therefore it cannot be ruled out that a better explanation could be obtained by replacing nervousness with excitement, or reaction to novelty, in the previous reasoning.

6.3 Verification of *E4UnityIntegration-MIT*

A secondary purpose of the study was to serve as on the field testing for the *E4UnityIntegration-MIT* plugin. Indeed, the plugin proved capable of handling the task of capturing participants' physiological data during the VR game.

The process was not without some slight troubles. The game had been modified with the behaviour of connecting automatically to the E4 when the main scene started; there were, however, two instances in which this failed to happen. Thankfully, in one of the cases this problem was detected in time, and the researcher stopped the experience at the beginning of the main scene and restarted the game to try again. Thus, only one participant had no E4 data recorded for them (this is one of the incidents referred to in section 5.3 that warranted exclusion from data analysis).

For comparison, the E4 also failed twice in the baseline recording stage. These measurements did not depend on *E4UnityIntegration-MIT* but rather on Empatica's own E4 realtime smartphone app. As such, the two incidences of *E4UnityIntegration-MIT* problems do not present an increase (nor decrease) with respect to Empatica's first party software.

The exact cause of *E4UnityIntegration-MIT*'s malfunctions is unknown. Further tests would need to be done after implementing some sort of logging utility in order to keep record of the events related to E4 status during a malfunction to enable a precise diagnosis.

Nevertheless, it is known that the error is not reliably reproducible, as evidenced by the fact that it did not occur with every participant. Errors that exhibit this sort of inconsistent behaviour are often related to network or connectivity issues (in this case, perhaps connection to the internet though WiFi, or the Bluetooth connection between the E4 and the headset), as such means of communication are fallible mediums. The internet access dependency of *E4UnityIntegration-MIT* is actually inherited from the Android E4 library itself, which requires authentication of the application with Empatica's servers in order to have permission to use a particular E4 device [1]. This renders the correct functioning of the plugin reliant on

[1] As already commented on in (section 4.2), the authentication requirement, which consists of having the application register the developer API key with Empatica's servers, is used to determine whether a particular program's developer has permission to connect to a given E4 device. This is a necessary feature to avoid potential privacy issues (i.e., having a malicious

the Empatica servers being online and accessible at the time of connection, which in general can be expected but never guaranteed.

Indeed, it is our suspicion that such an issue is at play in these situations. In one of the *E4UnityIntegration-MIT* malfunction instances the same participant had to have their baseline measures retaken because an E4 realtime failure happened as well (the first time around no data was stored on the cloud). This points to there having been some problem beyond the software, either on the E4 hardware (unlikely since the device worked correctly before and after this incident) or, more likely, on the network side of things.

Regardless of where specifically the problem lies, it cannot be ignored that the previously explained network reliance is a potential risk. Researchers (or any users of the plugin, more generally) ought to implement safeguards that impede progress in their Unity application in case connection fails (we remark that implementing such a feature would be relatively easy, utilizing the facilities the plugin provides). While this was not done for this study due to time constraints, such a safeguard would also automatically protect against other bugs of the plugin, should the preceding assessment of the likely root of the problem prove to be misguided.

In short, while there were incidents, issues like the ones mentioned can easily be (and should be) guarded against; and, in any case, use of *E4UnityIntegration-MIT* has proven to be no less reliable than Empatica's own solutions anyway. As such, we regard the plugin to be validated as fit for use, for the purpose of gathering physiological data from Empatica E4 for research or otherwise.

program that automatically connects to any E4 it finds, regardless of whether the device belongs to the developer of the program or not, and which can therefore spy on the physiological status of whoever is wearing it), which is essential for a health device.

Chapter 7

Conclusions

Sense of presence is an important factor affecting player experience. Just as one may study the relationships between psychophysiological measurements and PX, so too can one ask:

> How are players' physiological responses related to their sense of presence during a VR roleplaying game?

The study presented in this thesis project was an exploratory investigation on this matter through an experiment presented in Part III, which constitutes the project's experimental contribution.

In order to carry out the experiment, it became necessary to develop the integration for one of the measurement tools, the Empatica E4; the code ended up being constituted as *E4UnityIntegration-MIT*. This plugin was published as open source with the intention of providing a ready-made solution to the desire to integrate the E4 with Unity on Android. The study itself also provided a chance to validate the tool in a realistic experimental research setting while being used for one of its intended purposes, that being data gathering and storing for research. *E4UnityIntegration-MIT*, presented in Part II, alongside its verification in the study, make up the project's technical contribution.

All in all, we can say the project delivered on both accounts. Besides the overall success in piloting *E4UnityIntegration*, in chapter 6 the analysis of psychophysiological data exposes that, while no other relationship is statistically supported, there is a significant negative correlation between temperature and sense of presence. We attribute this to thermal discomfort, and surmise that participants who were too hot during the experience felt a diminished sense of presence.

We can hope that *E4UnityIntegration-MIT* is of service to the broader game research and development communities. On the other hand, the experimental results

give us ground to be more prescriptive than just wishful thinking.

Based on the findings, it seems that as participants' body temperature increased, their sense of presence decreased. This indicates that increased body temperature may undermine sense of presence during VR games. Future research should experimentally manipulate room temperature to elucidate if there is a causal relationship between the two variables.

Additionally, EDA did not achieve a significant positive correlation in this study, but it came relatively close. A study with a larger sample size could be conducted to determine whether there is smaller scale influence on sense of presence at play.

Finally, only a selection of all psychophysiological measurements was analysed in relation to presence. Other studies could also go in the direction of exploring how other measurements relate to presence.

In conclusion, work remains to be done on the open research question of the relationship between physiological measurements and sense of presence. Together with some findings of our own, we have reported a range of possible avenues of further inquiry and remain expectant of future developments in this exciting area of research.

Bibliography

[1] *Advanced research on human behavior*. URL: https://www.empatica.com/research/publications/.

[2] *An Android Plugin for E4 Wristband Integration in Unity*. URL: https://osf.io/v9whk/?view%5C_only=dc43354770044134a45c0a74c312514f.

[3] *Android plug-in types*. URL: https://docs.unity3d.com/Manual/android-plugin-types.html.

[4] *Android Studio*. URL: https://developer.android.com/studio.

[5] Stuart M Bender and Billy Sung. "Fright, attention, and joy while killing zombies in virtual reality: A psychophysiological analysis of VR user experience". In: *Psychology & Marketing* 38.6 (2021), pp. 937–947.

[6] Emily Brown and Paul Cairns. "A grounded investigation of game immersion". In: *CHI'04 extended abstracts on Human factors in computing systems*. 2004, pp. 1297–1300.

[7] Aaron Frederick Bulagang, James Mountstephens, and Jason Teo. "Multiclass emotion prediction using heart rate and virtual reality stimuli". In: *Journal of Big Data* 8 (2021), pp. 1–12.

[8] *Call for Participation in VR Research*. URL: https://www.reddit.com/r/mit/comments/13bvtot/call_for_participation_in_vr_research/.

[9] Rafael A Calvo et al. *The Oxford handbook of affective computing*. Oxford Library of Psychology, 2015.

[10] Michael Carroll, Ethan Osborne, and Caglar Yildirim. "Effects of VR gaming and game genre on player experience". In: *2019 IEEE Games, Entertainment, Media Conference (GEM)*. IEEE. 2019, pp. 1–6.

[11] Francesco Chiossi et al. "Virtual reality adaptation using electrodermal activity to support the user experience". In: *Big Data and Cognitive Computing* 6.2 (2022), p. 55.

[12] Mihaly Csikszentmihalyi. *Beyond boredom and anxiety*. Jossey-bass, 2000.

[13] *Data export and formatting from E4 connect*. URL: https://support.empatica.com/hc/en-us/articles/201608896-Data-export-and-formatting-from-E4-connect-.

[14] *Decoding wearable sensor signals - what to expect from your E4 Data*. URL: https://www.empatica.com/blog/decoding-wearable-sensor-signals-what-to-expect-from-your-e4-data.html.

[15] Anders Drachen et al. "Correlation between heart rate, electrodermal activity and player experience in first-person shooter games". In: *Proceedings of the 5th ACM SIGGRAPH Symposium on Video Games*. 2010, pp. 49–54.

[16] *E4 connect*. URL: https://e4.empatica.com/connect.

[17] *E4 for Developer*. URL: https://e4.empatica.com/connect/developer.php.

[18] *E4 link SDK for Android*. URL: https://developer.empatica.com/android-sdk-tutorial-100.html.

[19] *E4 streaming server*. URL: https://developer.empatica.com/windows-streaming-server.html.

[20] *E4 wristband*. URL: https://www.empatica.com/research/e4/.

[21] *E4 wristband for Developers*. URL: https://developer.empatica.com/.

[22] *E4 wristband technical specifications*. URL: https://support.empatica.com/hc/en-us/articles/202581999-E4-wristband-technical-specifications.

[23] *E4link Android SDK Javadoc*. URL: https://developer.empatica.com/empatica-android-sdk-javadoc.zip.

[24] *E4link Sample Project*. URL: https://github.com/empatica/e4link-sample-project-android.

[25] Darragh Egan et al. "An evaluation of Heart Rate and ElectroDermal Activity as an objective QoE evaluation method for immersive virtual reality environments". In: *2016 eighth international conference on quality of multimedia experience (QoMEX)*. IEEE. 2016, pp. 1–6.

[26] Sohye Lim and Byron Reeves. "Being in the game: Effects of avatar choice and point of view on psychophysiological responses during play". In: *Media psychology* 12.4 (2009), pp. 348–370.

[27] *Meta Quest*. URL: https://www.meta.com/quest/.

[28] *Meta Quest 2*. URL: https://www.meta.com/quest/products/quest-2/.

[29] Lennart E Nacke. "An introduction to physiological player metrics for evaluating games". In: *Game Analytics: Maximizing the value of player data* (2013), pp. 585–619.

[30] Lennart E Nacke and Craig A Lindley. "Affective ludology, flow and immersion in a first-person shooter: Measurement of player experience". In: *arXiv preprint arXiv:1004.0248* (2010).

[31] *Native Code expects "didUpdateOnWristStatus" in "EmpaDataDelegate"*. URL: https://github.com/empatica/e4link-sample-project-android/issues/2.

[32] *OkHttp*. URL: https://square.github.io/okhttp/.

[33] Karolien Poels, Yvonne AW de Kort, and Wijnand A IJsselsteijn. "D3. 3: Game Experience Questionnaire: development of a self-report measure to assess the psychological impact of digital games". In: (2007).

[34] Helena Polman. "The Impact of Changing Moods Based on Real-Time Biometric Measurements on Player Experience". In: *Gaming, Simulation and Innovations: Challenges and Opportunities: 52nd International Simulation and Gaming Association Conference, ISAGA 2021, Indore, India, September 6–10, 2021, Revised Selected Papers*. Springer. 2022, pp. 171–181.

[35] *Quest 2 Navigating in VR*. URL: https://www.youtube.com/watch?v=gKkwfy9GueQ.

[36] James A Russell. "A circumplex model of affect." In: *Journal of personality and social psychology* 39.6 (1980), p. 1161.

[37] Thomas Schubert, Frank Friedmann, and Holger Regenbrecht. "The experience of presence: Factor analytic insights". In: *Presence: Teleoperators & Virtual Environments* 10.3 (2001), pp. 266–281.

[38] Soo Young Sim et al. "Estimation of thermal sensation based on wrist skin temperatures". In: *Sensors* 16.4 (2016), p. 420.

[39] Elton Sarmanho Siqueira et al. "An automated approach to estimate player experience in game events from psychophysiological data". In: *Multimedia Tools and Applications* (2022), pp. 1–32.

[40] *Tuples vs System.Tuple*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples#tuples-vs-systemtuple.

[41] *Unity*. URL: https://unity.com/.

[42] *View logs with Logcat*. URL: https://developer.android.com/studio/debug/logcat.

[43] Caglar Yildirim and D Fox Harrell. "On the Plane: A Roleplaying Game for Simulating Ingroup-Outgroup Biases in Virtual Reality". In: *2022 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*. IEEE. 2022, pp. 207–209.

[44] Caglar Yildirim et al. "Video game user experience: to VR, or not to VR?" In: *2018 IEEE Games, Entertainment, Media Conference (GEM)*. IEEE. 2018, pp. 1–9.

Chapter A

E4UnityIntegration-MIT plugin

The plugin is freely available online at [2].

A.1 Source Code

A.1.1 `AndroidManifest.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.unity3d.player"
    xmlns:tools="http://schemas.android.com/tools">
    <application>
        <activity android:name="com.unity3d.player.UnityPlayerActivity"
                  android:theme="@style/UnityThemeSelector">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <meta-data android:name="unityplayer.UnityActivity"
            ↪  android:value="true" />
        </activity>
    </application>
    <!-- E4link -->
    <uses-permission android:name="android.permission.BLUETOOTH" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-feature
        android:name="android.hardware.bluetooth_le"
```

```
        android:required="true" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
</manifest>
```

Listing 11: `AndroidManifest.xml`

A.1.2 `mainTemplate.gradle`

```
apply plugin: 'com.android.library'
**APPLY_PLUGINS**

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.squareup.okhttp:okhttp:2.7.5'
**DEPS**}

android {
    compileSdkVersion **APIVERSION**
    buildToolsVersion '**BUILDTOOLS**'

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }

    defaultConfig {
        minSdkVersion **MINSDKVERSION**
        targetSdkVersion **TARGETSDKVERSION**
        ndk {
            abiFilters **ABIFILTERS**
        }
        versionCode **VERSIONCODE**
        versionName '**VERSIONNAME**'
        consumerProguardFiles 'proguard-unity.txt'**USER_PROGUARD**
    }

    lintOptions {
        abortOnError false
    }

    aaptOptions {
        noCompress = **BUILTIN_NOCOMPRESS** + unityStreamingAssets.tokenize(',
        ↪  ')
```

```
35          ignoreAssetsPattern =
            ↪ "!.svn:!.git:!.ds_store:!*.scc:.*:!CVS:!thumbs.db:!picasa.ini:!*~"
36      }**PACKAGING_OPTIONS**
37  }**REPOSITORIES**
38  **IL_CPP_BUILD_SETUP**
39  **SOURCE_BUILD_SETUP**
40  **EXTERNAL_SOURCES**
```

Listing 12: `mainTemplate.gradle`

A.1.3 UnityE4.cs

```
1   using UnityEngine;
2   using UnityEngine.Android;
3   using System.Collections;
4   using System.Collections.Concurrent;
5   using System.IO;
6   using System.Text;
7   using System;
8
9   public class UnityE4 : MonoBehaviour
10  {
11      // Data logging section
12
13      private enum LoggerState
14      {
15          Initial,
16          Start,
17          Logging,
18          Stop,
19          DidWrite,
20          FailedWrite
21      }
22
23      private LoggerState loggerState = LoggerState.Initial;
24
25      // Tuple class is reference type so it is thread-safe to keep track of the
    ↪   last one
26      public Tuple<int, int, int, double> accLast = new Tuple<int, int, int,
    ↪   double>(0, 0, 0, 0.0d);
27      public Tuple<float, double> batteryLast = new Tuple<float, double>(0.0f,
    ↪   0.0d);
28      public Tuple<float, double> bvpLast = new Tuple<float, double>(0.0f, 0.0d);
29      public Tuple<float, double> gsrLast = new Tuple<float, double>(0.0f, 0.0d);
30      public Tuple<float, double> ibiLast = new Tuple<float, double>(0.0f, 0.0d);
```

```
31      public Tuple<float, double> tempLast = new Tuple<float, double>(0.0f, 0.0d);
32
33      private StringBuilder csvString;
34      private StreamWriter file;
35
36      private float currentSecond = 0.0f;
37
38      private IEnumerator E4DataLogger()
39      {
40          // Should now be loggerState == LoggerState.Start
41          GenerateCsvHeader();
42          loggerState = LoggerState.Logging;
43          while (loggerState == LoggerState.Logging)
44          {
45              LogConstruction();
46              yield return new WaitForSecondsRealtime(0.25f);
47              currentSecond += 0.25f;
48          }
49          // Should now be loggerState == LoggerState.Stop
50          if (WriteToFile())
51          {
52              loggerState = LoggerState.DidWrite;
53          }
54          else
55          {
56              loggerState = LoggerState.FailedWrite;
57          }
58      }
59
60      private void GenerateCsvHeader()
61      {
62          csvString = new StringBuilder();
63
64          string[] header =
65          {
66                      "Time",
67                      "AccX", "AccY", "AccZ", "AccT",
68                      "Bvp", "BvpT",
69                      "Gsr", "GsrT",
70                      "Ibi", "IbiT",
71                      "Temp", "TempT"
72          };
73
74          foreach (string s in header)
75          {
76              csvString.Append(s + ", ");
77          }
```

```
78          csvString.Append("\n");
79
80      }
81
82      private void LogConstruction()
83      {
84          Tuple<int, int, int, double> acc = accLast;
85          Tuple<float, double> battery = batteryLast;
86          Tuple<float, double> bvp = bvpLast;
87          Tuple<float, double> gsr = gsrLast;
88          Tuple<float, double> ibi = ibiLast;
89          Tuple<float, double> temp = tempLast;
90
91          csvString.AppendFormat("{0,5:F2}, " +
92              "{1}, {2}, {3}, {4,5:F2}, " +
93              "{5}, {6,5:F2}, " +
94              "{7}, {8,5:F2}, " +
95              "{9}, {10,5:F2}, " +
96              "{11}, {12,5:F2}\n",
97              currentSecond,
98              acc.Item1, acc.Item2, acc.Item3, acc.Item4 % 1000.0f,
99              bvp.Item1, bvp.Item2 % 1000.0f,
100             gsr.Item1, gsr.Item2 % 1000.0f,
101             ibi.Item1, ibi.Item2 % 1000.0f,
102             temp.Item1, temp.Item2 % 1000.0f);
103     }
104
105     private bool WriteToFile()
106     {
107         string PID = "P000";
108
109         string fname = PID + ".csv";
110         string path = Path.Combine(Application.persistentDataPath, fname);
111
112         try
113         {
114             file = new StreamWriter(path);
115             file.WriteLine(csvString.ToString());
116             file.Close();
117             return true;
118         }
119         catch
120         {
121             return false;
122         }
123
124     }
```

```
125
126     private void UpdateLogger()
127     {
128         if (loggerState == LoggerState.Start)
129         {
130             StartCoroutine(E4DataLogger());
131         }
132     }
133
134     // Delegate callback section
135
136     // Note: delegate callbacks cannot use Unity API calls because they don't
    ↪  run on the main Unity thread
137
138     class UnityE4linkDataDelegateCallback : AndroidJavaProxy
139     {
140         private readonly UnityE4 unityE4;
141
142         public UnityE4linkDataDelegateCallback(UnityE4 unityE4) :
    ↪  base("edu.mit.virtuality.unityE4link.UnityE4link$UnityE4linkDataDelegate")
143         {
144             this.unityE4 = unityE4;
145         }
146
147         void didReceiveAcceleration(int x, int y, int z, double timestamp)
148         {
149             Tuple<int, int, int, double> accTuple = new Tuple<int, int, int,
    ↪  double>(x, y, z, timestamp);
150             unityE4.accLast = accTuple;
151         }
152
153         void didReceiveBatteryLevel(float battery, double timestamp)
154         {
155             Tuple<float, double> batteryTuple = new Tuple<float,
    ↪  double>(battery, timestamp);
156             unityE4.batteryLast = batteryTuple;
157         }
158
159         void didReceiveBVP(float bvp, double timestamp)
160         {
161             Tuple<float, double> bvpTuple = new Tuple<float, double>(bvp,
    ↪  timestamp);
162             unityE4.bvpLast = bvpTuple;
163         }
164
165         void didReceiveGSR(float gsr, double timestamp)
166         {
```

```
167             Tuple<float, double> gsrTuple = new Tuple<float, double>(gsr,
    ↪ timestamp);
168             unityE4.gsrLast= gsrTuple;
169         }
170 
171         void didReceiveIBI(float ibi, double timestamp)
172         {
173             Tuple<float, double> ibiTuple = new Tuple<float, double>(ibi,
    ↪ timestamp);
174             unityE4.ibiLast = ibiTuple;
175         }
176 
177         void didReceiveTemperature(float temp, double timestamp)
178         {
179             Tuple<float, double> tempTuple = new Tuple<float, double>(temp,
    ↪ timestamp);
180             unityE4.tempLast = tempTuple;
181         }
182 
183         // Following method does not appear in E4 link documentation
184         void didReceiveTag(double timestamp)
185         {
186         }
187     }
188 
189     class UnityE4linkStatusDelegateCallback : AndroidJavaProxy
190     {
191         private readonly UnityE4 unityE4;
192 
193         public UnityE4linkStatusDelegateCallback(UnityE4 unityE4) :
    ↪ base("edu.mit.virtuality.unityE4link.UnityE4link$UnityE4linkStatusDelegate")
194         {
195             this.unityE4 = unityE4;
196         }
197 
198         void didUpdateStatus(string status)
199         {
200             if (status == "CONNECTED")
201             {
202                 unityE4.loggerState = LoggerState.Start;
203             }
204             else if (status == "DISCONNECTED")
205             {
206                 unityE4.loggerState = LoggerState.Stop;
207             }
208         }
209 
```

```
210         void didEstablishConnection()
211         {
212         }
213 
214         void didUpdateSensorStatus(string status, string type)
215         {
216         }
217 
218         void didDiscoverDevice(string deviceName, int rssi, bool allowed)
219         {
220         }
221 
222         void didFailedScanning(string error)
223         {
224         }
225 
226         void didRequestEnableBluetooth()
227         {
228         }
229 
230         void bluetoothStateChanged(bool isBluetoothOn)
231         {
232         }
233 
234         void didUpdateOnWristStatus(string status)
235         {
236         }
237     }
238 
239 
240     private readonly string EMPATICA_API_KEY = "YOUR_API_KEY";
241 
242     private UnityE4linkDataDelegateCallback unityE4linkDataDelegateCallback;
243     private UnityE4linkStatusDelegateCallback unityE4linkStatusDelegateCallback;
244     private AndroidJavaObject unityE4link;
245 
246 
247     // Permissions section
248 
249     private bool HasUnityE4linkPermissions()
250     {
251         return Permission.HasUserAuthorizedPermission(Permission.FineLocation);
252     }
253     // Call only if you lack the necessary permissions
254     private void RequestUnityE4linkPermissions()
255     {
256         PermissionCallbacks permissionCallbacks = new PermissionCallbacks();
```

```
257         permissionCallbacks.PermissionGranted += (string s) => {
    ↪ InitUnityE4link(); };
258         // FineLocation is critical, so always request again.
259         // However, not google's recommendation on how to handle user denying
260         permissionCallbacks.PermissionDenied += (string s) => {
    ↪ RequestUnityE4linkPermissions(); };
261         Permission.RequestUserPermission(Permission.FineLocation,
    ↪ permissionCallbacks);
262     }
263     // Call only if you have the necessary permissions
264     private void InitUnityE4link()
265     {
266         unityE4link = new
    ↪ AndroidJavaObject("edu.mit.virtuality.unityE4link.UnityE4link",
    ↪ unityE4linkDataDelegateCallback, unityE4linkStatusDelegateCallback,
    ↪ EMPATICA_API_KEY);
267     }
268
269
270     void Start()
271     {
272         unityE4linkDataDelegateCallback = new
    ↪ UnityE4linkDataDelegateCallback(this);
273         unityE4linkStatusDelegateCallback = new
    ↪ UnityE4linkStatusDelegateCallback(this);
274
275         // Check permissions and initialize unityE4link
276         if (HasUnityE4linkPermissions())
277         {
278             InitUnityE4link();
279         }
280         else
281         {
282             // When granted will initialize unityE4link
283             RequestUnityE4linkPermissions();
284         }
285     }
286
287     void Update()
288     {
289         // Stop logging after 40 seconds
290         if (currentSecond >= 40.0f)
291         {
292             loggerState = LoggerState.Stop;
293         }
294         UpdateLogger();
295     }
```

```
296 }
```

Listing 13: UnityE4.cs

A.1.4 UnityE4link.java

```
1  package edu.mit.virtuality.unityE4link;
2
3  import android.Manifest;
4  import android.app.Activity;
5  import android.app.AlertDialog;
6  import android.bluetooth.BluetoothAdapter;
7  import android.bluetooth.le.ScanCallback;
8  import android.content.DialogInterface;
9  import android.content.Intent;
10 import android.content.pm.PackageManager;
11 import android.net.Uri;
12 import android.os.Bundle;
13 import android.provider.Settings;
14 import android.text.TextUtils;
15 import android.util.Log;
16 import android.view.View;
17 import android.widget.Button;
18 import android.widget.LinearLayout;
19 import android.widget.TextView;
20 import android.widget.Toast;
21
22 import com.empatica.empalink.ConnectionNotAllowedException;
23 import com.empatica.empalink.EmpaDeviceManager;
24 import com.empatica.empalink.EmpaticaDevice;
25 import com.empatica.empalink.config.EmpaSensorStatus;
26 import com.empatica.empalink.config.EmpaSensorType;
27 import com.empatica.empalink.config.EmpaStatus;
28 import com.empatica.empalink.delegate.EmpaDataDelegate;
29 import com.empatica.empalink.delegate.EmpaStatusDelegate;
30
31 import com.unity3d.player.UnityPlayer;
32
33
34 public class UnityE4link {
35     // Implement these interfaces in a Unity C# script to receive the E4's data
      ↪  and status updates
36     interface UnityE4linkDataDelegate extends EmpaDataDelegate { }
37     interface UnityE4linkStatusDelegate {
38         void didUpdateStatus(String status);
```

```
39          void didEstablishConnection();
40          void didUpdateSensorStatus(String status, String type);
41          void didDiscoverDevice(String deviceName, int rssi, boolean allowed);
42          void didFailedScanning(String error);
43          void didRequestEnableBluetooth();
44          void bluetoothStateChanged(boolean isBluetoothOn);
45          void didUpdateOnWristStatus(String status);
46      }
47
48      // Internal class, essential to work around a bug in the Empatica's E4 link
        ↪  library
49      // Passes callbacks through to the Unity C# script's
50      // Additionally, default behaviour of connecting to the first E4 found is
        ↪  implemented here
51      public class EmpaDelegate implements EmpaDataDelegate, EmpaStatusDelegate {
52          private final UnityE4linkDataDelegate unityE4linkDataDelegate;
53          private final UnityE4linkStatusDelegate unityE4linkStatusDelegate;
54
55          public EmpaDelegate(UnityE4linkDataDelegate unityE4linkDataDelegate,
            ↪  UnityE4linkStatusDelegate unityE4linkStatusDelegate) {
56              this.unityE4linkDataDelegate = unityE4linkDataDelegate;
57              this.unityE4linkStatusDelegate = unityE4linkStatusDelegate;
58          }
59
60
61          // EmpaDataDelegate methods
62          public void didReceiveAcceleration(int x, int y, int z, double
            ↪  timestamp) {
63              unityE4linkDataDelegate.didReceiveAcceleration(x, y, z, timestamp);
64          }
65
66          public void didReceiveBatteryLevel(float battery, double timestamp) {
67              unityE4linkDataDelegate.didReceiveBatteryLevel(battery, timestamp);
68          }
69
70          public void didReceiveBVP(float bvp, double timestamp) {
71              unityE4linkDataDelegate.didReceiveBVP(bvp, timestamp);
72          }
73
74          public void didReceiveGSR(float gsr, double timestamp) {
75              unityE4linkDataDelegate.didReceiveGSR(gsr, timestamp);
76          }
77
78          public void didReceiveIBI(float ibi, double timestamp) {
79              unityE4linkDataDelegate.didReceiveIBI(ibi, timestamp);
80          }
81
```

```
82          public void didReceiveTemperature(float temp, double timestamp) {
83              unityE4linkDataDelegate.didReceiveTemperature(temp, timestamp);
84          }
85
86          // Following method does not appear in E4 link documentation
87          public void didReceiveTag(double timestamp) {
88              unityE4linkDataDelegate.didReceiveTag(timestamp);
89          }
90
91
92          private String toStringEmpaSensorStatus(@EmpaSensorStatus int status) {
93              switch (status) {
94                  case EmpaSensorStatus.NOT_ON_WRIST:
95                      return "NOT_ON_WRIST";
96                  case EmpaSensorStatus.ON_WRIST:
97                      return "ON_WRIST";
98                  case EmpaSensorStatus.DEAD:
99                      return "DEAD";
100                 default:
101                     return "UNKNOWN_STATUS";
102             }
103         }
104
105         // EmpaStatusDelegate methods
106         public void didUpdateStatus(EmpaStatus status) {
107             switch (status) {
108                 case INITIAL:
109                     break;
110                 case READY: {
111                     // Start scanning
112                     deviceManager.startScanning();
113                 } break;
114                 case CONNECTED:
115                     break;
116                 case DISCONNECTED:
117                     break;
118                 case CONNECTING:
119                     break;
120                 case DISCONNECTING:
121                     break;
122                 case DISCOVERING:
123                     break;
124             }
125
126             unityE4linkStatusDelegate.didUpdateStatus(status.toString());
127         }
128
```

```
129         public void didEstablishConnection() {
130             unityE4linkStatusDelegate.didEstablishConnection();
131         }
132 
133         public void didUpdateSensorStatus(@EmpaSensorStatus int status,
            ↪ EmpaSensorType type) {
134             didUpdateOnWristStatus(status);
135 
136             unityE4linkStatusDelegate.didUpdateSensorStatus(toStringEmpaSensorStatus(status),
                ↪ type.toString());
137         }
138 
139         public void didDiscoverDevice(EmpaticaDevice bluetoothDevice, String
            ↪ deviceName, int rssi, boolean allowed) {
140             if (allowed) {
141                 // Stop scanning. The first allowed device will do.
142                 deviceManager.stopScanning();
143                 try {
144                     // Connect to the device
145                     deviceManager.connectDevice(bluetoothDevice);
146                 } catch (ConnectionNotAllowedException e) {
147                     // This should happen only if you try to connect when
                        ↪ allowed == false.
148                 }
149             }
150 
151             unityE4linkStatusDelegate.didDiscoverDevice(deviceName, rssi,
                ↪ allowed);
152         }
153 
154         public void didFailedScanning(int errorCode) {
155             String s;
156             switch (errorCode) {
157                 case ScanCallback.SCAN_FAILED_ALREADY_STARTED:
158                     s = "Scan failed: a BLE scan with the same settings is
                        ↪ already started by the app";
159                     break;
160                 case ScanCallback.SCAN_FAILED_APPLICATION_REGISTRATION_FAILED:
161                     s = "Scan failed: app cannot be registered";
162                     break;
163                 case ScanCallback.SCAN_FAILED_FEATURE_UNSUPPORTED:
164                     s = "Scan failed: power optimized scan feature is not
                        ↪ supported";
165                     break;
166                 case ScanCallback.SCAN_FAILED_INTERNAL_ERROR:
167                     s = "Scan failed: internal error";
168                     break;
```

```
169                 default:
170                     s = "Scan failed with unknown error (errorCode=" + errorCode
                        ↪  + ")";
171                     break;
172             }
173
174             unityE4linkStatusDelegate.didFailedScanning(s);
175         }
176
177         public void didRequestEnableBluetooth() {
178             // Requesting enabling bluetooth to the user can be implemented here
179
180             unityE4linkStatusDelegate.didRequestEnableBluetooth();
181         }
182
183         public void bluetoothStateChanged() {
184             // E4link detected a bluetooth adapter change
185             boolean isBluetoothOn =
                ↪  BluetoothAdapter.getDefaultAdapter().isEnabled();
186             unityE4linkStatusDelegate.bluetoothStateChanged(isBluetoothOn);
187
188         }
189
190         public void didUpdateOnWristStatus(@EmpaSensorStatus final int status) {
191             unityE4linkStatusDelegate.didUpdateOnWristStatus(toStringEmpaSensorStatus(status));
192         }
193     }
194
195
196     private final EmpaDelegate empaDelegate;
197     private final EmpaDeviceManager deviceManager;
198
199     public UnityE4link(UnityE4linkDataDelegate unityE4linkDataDelegate,
        ↪  UnityE4linkStatusDelegate unityE4linkStatusDelegate, String
        ↪  empaticaApiKey) {
200         Activity activity = UnityPlayer.currentActivity;
201
202         empaDelegate = new EmpaDelegate(unityE4linkDataDelegate,
            ↪  unityE4linkStatusDelegate);
203         deviceManager = new EmpaDeviceManager(activity.getApplicationContext(),
            ↪  empaDelegate, empaDelegate);
204
205         deviceManager.authenticateWithAPIKey(empaticaApiKey);
206     }
207 }
```

Listing 14: `UnityE4link.java`

A.2 Installation Instructions

1. Import the `AndroidManifest.xml` and `mainTemplate.gradle` files.

   To do this, enable Custom Main Manifest and Custom Main Gradle Template in the settings (Build Settings → Player Settings → Publishing Settings tab → Under the Build header). This creates some default files with the same name in the `Assets/Plugins/Android` folder, which can be replaced with the supplied ones.

   If a custom main manifest or main gradle template was already in use, you can merge the relevant part of either (the permissions and feature below the comment, in the case of `AndroidManifest.xml`, or the `okhttp:2.7.5` dependency in the case of `mainTemplate.gradle`) into the existing ones.

   Note: the `AndroidManifest.xml` and `mainTemplate.gradle` files must be in the `Assets/Plugins/Android` directory to work.

2. Import the *E4link for Android* library (`E4link.aar`).

   Note: the file must be in the `Assets/Plugins/Android` directory to work.

3. Import the the `UnityE4.cs` script wherever, and the `UnityE4link.java` file into `Assets/Plugins/Android`. The `UnityE4.cs` script receives all callbacks (data, status) from the library and handles writing the data to a file.

   `UnityE4.cs` can be attached to an empty GameObject. Replace the string in `private readonly string EMPATICA_API_KEY = "YOUR_API_KEY";` with a valid developer API key to have the plugin begin functioning; these steps suffice to trigger the default behaviour (asking for Bluetooth permissions if not possessed, connecting to the first E4 encountered, and recording data to a file).

   Note: the `UnityE4link.java` file must be in the `Assets/Plugins/Android` directory to work.