FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Automation of the Virtual Jump Simulator

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

BSc. Juri Berlanda
Matrikelnummer 00926383

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Horst Eidenberger

Wien, 7 März, 2019

Juri Berlanda	Horst Eidenberger

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Automation of the Virtual Jump Simulator

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Media Informatics and Visual Computing

by

BSc. Juri Berlanda
Registration Number 00926383

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Horst Eidenberger

Vienna, 7th March, 2019

Juri Berlanda	Horst Eidenberger

Erklärung zur Verfassung der Arbeit

BSc. Juri Berlanda
Hasengasse 27/20
1100 Wien

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

Wien, 7 März, 2019

Juri Berlanda

Acknowledgements

This thesis would not have been possible without a vast number of persons. First and foremost I'd like to thank the advisor of this thesis, Horst Eidenberger, who not only allowed me to work on this topic, but also managed to preserve a very pleasant working environment throughout the project, even in times of stress and nearing deadlines. Additionally, I'd like to thank him for the remarkable amount of both patience and empathy he presented towards me.

Next in line has to be my mother, Elisabeth Rassele, who always supported me throughout my studies, not only financially, but also morally. She always believed in me, even in times when I myself started doubting. Without her support this thesis would have never been written.

A special thank also goes to the other team members. Foremost would be Florin Hillebrand and Bela Eckermann, who were of great help for me and an invaluable source of inspiration, know-what, and know-how.

Not part of the team, but still worth a special thank is Gordan Savičić. He provided the designs and a prototype of what became to be the Dimmer device. He gave quick and useful answers to my questions, and all that even though he had no horse in the race.

Throughout the project I was standing on the shoulders of giants, namely the members and contributors of all the open source soft- and hardware projects used in this project. Naming them all is mere impossible, but having at disposal such things as the Linux, Git, Yocto, SocketIO, Python, and Flask projects on the software side and the Raspberry Pi, and Arduino projects on the hardware side is of immeasurable value. I'd also like to expand my thanks to the open source community as a whole, who successfully manages to show that even in an ever more profit oriented time and industry, a group of volunteers is able to produce sheer greatness without the need for monetary incentives.

Finally, I'd like to thank my girlfriend, Sophie Herrmann for her moral support throughout the becoming of this thesis. Having her was an illimitable benefit, especially during the darker times.

Kurzfassung

Diese Diplomarbeit stellt einen Ansatz vor, mit welchem zusätzliche Sinnesreize in eine Computergrafikanwendung mit Fokus auf Virtual Reality, genauer dem *Virtual Jump Simulator* (kurz *Jumpcube*), integriert werden können. Da keine der existierenden Lösungen unseren Ansprüchen genügte, haben wir uns dafür entschieden, eine offene, modulare und skalierbare Plattform zu entwickeln, welche Geräte zur Sinnesreizerzeugung mit Fokus auf Virtual Reality ermöglicht. Um die nötigen Kosten und die Entwicklungszeit niedrig zu halten, haben wir, wo immer möglich, auf bestehende Hardwarestandartkomponenten zurückgegriffen. Aus dem selben Grund haben wir für die Netzwerkkommunikation auf bestehende Webtechnologie, namentlich JSON und das SockteIO Protokoll aufgebaut. Als Programmiersprache für die Logik der einzelnen Geräten fiel unsere Wahl auf Python. Es wurde gezeigt, dass solch ein System sowohl flexibel, als auch einfach zu implementieren und zu warten ist. Es wurde weiters gezeigt, dass die erhöhte Latenz, welche durch den Einsatz einer Hochsprache wie Python einerseits und nicht echtzeitfähiger Netzwerkkommunikation andererseits entsteht, im Rahmen von Computergrafik vernachlässigbar ist. Um den Einfluss sensorischer Reize auf den/die BenutzerIn zu testen, wurde sowohl eine quantitative, als auch eine qualitative Studie durchgeführt. Beide zeigen, dass zusätzliche sensorische Reize in Virtual Reality von dem/der BenutzerIn wahrgenommen werden und einen positiven Einfluss auf die User Experience haben, besonders auf den empfundenen Grad von Präsenz.

Abstract

This thesis presents an approach for integrating multi-sensory feedback with a computer graphics environment designed for use with Virtual Reality, namely the *Virtual Jump Simulator*, or *Jumpcube* for short. Since none of the existing solutions would meet our requirements, we decided to build an open, modular, and scalable platform for creating multi-sensory feedback devices to be used in Virtual Reality. To keep development costs and time low we used mainly off-the-shelf hardware components. For the same reason we also opted for a communication backbone based on Ethernet, modern Web technologies, namely JSON and the SocketIO protocol, and Python as the programming language for the logic running on the multi-sensory feedback devices. We demonstrated, that such a communication backbone is flexible and both easy to implement and to maintain. Further, we showed that the additional latency introduced by the use of high-level programming languages and non-real time capable communication is negligible in the scope of a computer graphics environment. To determine the impact the system has on the user we conducted both a quantitative and a qualitative experiment. Both showed that multi-sensory feedback in a Virtual Reality environment is noticed by the vast majority of the users and has a positive influence on the overall user experience, in particular on the degree of presence reached by the test subject.

Contents

CHAPTER 1

Introduction

1.1 Goals

Virtual Reality, even though having been in existence for a long time, made significant steps towards mainstream availability during the last five years. The author sees the follwing main reasons for this progress:

- The smartphone boom made high resolution displays with low screen sizes available at affordable prices as they became mass-produced goods
- Because of technical evolution consumer PC and gaming console hardware is today able to cope with the additional performance requirements Virtual Reality demands

This paved the way for mass market Virtual Reality headsets, the first of which, the "Oculus Rift DK1", was announced by Oculus VR in April 2012. Quite rapidly the idea got momentum, leading to a massive media hype on the topic.

Fast forward to 2016: Following the media hype tech giants Valve and Sony Entertainment announced and launched to market their own solutions for mainstream Virtual Reality, being "SteamVR" and "Playstation VR" respectively. American research and advisory firm Gartner declared in its 2016 "Hype Cycle for Emerging Technologies" that Virtual Reality is past the "Trough of Disillusionment" and on its way up the "Slope of Enlightenment", though still anticipating five to ten years to mainstream adoption (or in their terms the "Plateau of Productivity") [Gar16], a timespan they reduced to two to five years only one year after in their 2017 edition of "Hype Cycle for Emerging Technologies" [Gar17].

In the meantime at TU Wien the project "Virtual Jump Simulator", referred to as Jumpcube in the context of this document, was introduced. The idea was to develop a Virtual Reality system, which allows to simulate a parachute jump as accurately as

possible. Soon it went obvious that visual components would not be enough to achieve the main goal of any Virtual Reality system: presence through immersion [BM07]. To stick to the terminology proposed by Slater [Sla03] we consider immersion to be the objectively quantifiable properties of a system, like e.g. number of pixels or computing power, while the term presence points to the subjective experience of the user to not perceive the world being presented to him/her through the Virtual Reality headset as virtual or synthetic, but feel as if he/she is actually within that world. Please notice, that this does not mean that the main goal is to make the world as realistic as possible, as one can also feel present in e.g. a mystical world of elves and dragons. Still, in the best case the user would completely forget about reality around him/her and replace it with the virtual world (see Section 2.1.4 for further details).

To address more senses then just visual and aural, the Jumpcube needed some kind of environment control system, allowing for haptic feedback, g-force simulation or even olfactory elements to be included. Therefore the system needs to be very flexible and easily extensible for the usage with different peripherals, for example but not limited to fans, valves and motors. This should not only improve the user's experience, but also reduce the risk of experiencing simulator sickness. The latter is a phenomenon assumed to be closely related to sea sickness or car sickness [LD05]. It arises from senses contradicting each other, for example: Take a car passenger. The visual sense experiences no acceleration because some objects (e.g. the car's interior) are not moving at all or at constant speed but the inner ear feels acceleration (e.g. the car takes a turn). The brain cannot make sense of these obviously contradictory information and as a consequence the person starts feeling nauseous (see Section 2.1.5 for further details).

As a consequence the environment control system does not necessarily need to simulate every little detail of the virtual environment in a physically correct way. The goal is to provide a "good enough" approximation to support and emphasize visual experience, and before all the environment control system must not create sensory information contradicting the visual experience.

The Jumpcube is meant to offer a variety of different experiences, some of them being interactive, based on different technologies spanning from computer graphics to 360° video. Hence the environment control system must offer an interface accessible by a variety of different applications. As a best case scenario the environment control system would offer an API based on one or more widely adopted communication protocols, which allow it to seamlessly integrate with a wide range of software. If this is not possible some communication bridges need to be implemented. Please notice, that this interoperability constraint applies to the full communication stack, which apart from software and protocols, also includes the hardware (e.g. cables).

Due to the previously mentioned interactivity of offered experiences, the impact on the environment varies with the user's decisions and hence the environment control system has to be able to react to changes on demand. In addition the latency between an event being sent from any software or hardware component of the Jumpcube and the reaction by the environment control system shall not be noticeable by the user. In the best case

scenario the environment control system's latency is within one frame. Since the current Virtual Reality headsets offer a display fresh rate of 90 Hz, this boils down to the latency being lower then 11.1 ms.

Last but not least - wherever possible - free and open source hardware and software shall be used for creating the environment control system. This is before all a moral decision by the author but also offers many advantages in various aspects of the project. For further information on this topic see Section 2.4.1. As a consequence of the deep beliefs of the author in the free and open source mentality software and custom hardware originating from this project shall be put under a copyleft license and made available for the public.

1.2 Motivation

Similar to Virtual Reality, automation systems have existed for a long time, from rudimentary ones such as timer switches to highly complex systems, like fully automated production lines. And these systems too managed to spread to end customers during the last years, for example in form of home automation kits.

Such systems come in a virtually endless number of forms, from wired to wireless, from short range to long ranged, with or without hard realtime capabilities, message or status based, open or proprietary, only to mention a few key characteristics. Discussing them all is out of scope for this thesis, but some candidates which may be applicable for use in the Jumpcube are analyzed and explained in more detail in Section 2.3.2.

Yet, browsing through the literature problems arise: Some systems are optimized for video games, like servo bases for car seats targeting racing simulation enthusiasts. Hence, they offer out of the box integration with game engines at least to some extend. Being optimized for gaming experiences, they also claim to have acceptable latency. But they turn out to be very expensive and hardly (if even) extensible and customizable. Other systems like home automation kits are available at reasonable costs and are easily extensible, but hardly customizable and mostly restricted to wireless communication, which is a risk factor in unknown conditions like exhibitions with - taken into account as good as every single visitor carries a smartphone with him/her - a huge number of WiFi capable devices, heavily congesting both the 2.4GHz and the 5GHz band, which turns out to be the bands where modern and reasonably flexible wireless automation protocols operate in. Some systems like for example CAN have very good response time and realtime capabilities, but tend to be rather costly [1].

Others, like cheap remote controlled power switches are available at more reasonable costs, but are very slow to react. As a short example: a delay of half a second is completely

[1] The products referred to are IXXAT USB-to-CAN V2 listing at Conrad for 294,52 € (https://www.conrad.de/de/can-umsetzer-usb-can-bus-ixxat-101028112001-betriebsspannung-5-vdc-1386382.html), and B&B SmartWorx, Inc. PCIE-1680-AE listing at DigiKey for 345.00 $ (https://www.digikey.com/product-detail/en/b-b-smartworx-inc/PCIE-1680-AE/PCIE-1680-AE-ND/7426509). Prices retrieved at 30th April, 2018, 14:21.

acceptable for a light switch, but for simulating multi-sensory experiences in a Virtual Reality environment it is not.

In addition a homogeneous system is to be preferred as it eases the build up and tear down of the Jumpcube's bearing structure, which - taken it travels from time to time - is a use case to be addressed. Having multiple systems acting side by side also introduces the need of either a communication gateway between one and another or multiple different APIs, which all need to be implemented by all supported game engines, as well as the control panel.

So it soon became clear all of the existing systems bear at least some showstopper in the Jumpcube's context. Combining different systems with different strengths is not an acceptable way to go too. Yet this analysis also revealed what these systems are at heart: a processor and some sensors and/or actors, paired with some kind of connectivity. The base concept being simple, and the individual parts being available off-the-shelf at reasonable pricing led to the decision to design the whole system from scratch, relying on as much off-the-shelf hardware and existing software as possible.

1.3 Overview

In the context of the Jumpcube a wide variety of components needs to be controlled, for example but not limited to 220V AC fans, servo motors, and water valves for simulating wind, g-forces and clouds respectively. Also it must be taken into account, that at any time a new component needs to be integrated with the existing system.

All of these actors need to be integrated with the Virtual Reality experiences created by students in other projects accompanying the one this thesis refers to. Due to the multitude of different experiences the need for a clear and universal API arises. Additionally, in the context of VR realtime capabilities are needed at least to some extend. Both visual and otherwise experienced sensory input have to be synchronous for them to integrate with each other.

In the next chapter some past work will be presented starting with an analysis on what Virtual Reality is in Section 2.1.1. This also includes the principles on how it works from both technical and anatomic perspective and what the state of the art currently has to offer. Next a tour will be taken into the human cognitive aspects of this technology, showing how shortcomings in the human cognitive apparatus can be exploited to create an out of body experience in Section 2.1.4. Closing the section on Virtual Reality in 2.1.5 the topic will turn to what can go wrong if the human cognitive apparatus is not addressed properly. The socalled simulator sickness will be explained and also some techniques how to avoid or at least reduce its effect will be presented.

The next section will then lead over to the more technical part. The concepts of game engines will be discussed and it will be shown why they became the backbone of the video game industry. The interested reader can find this discussion in Section 2.2.

Eventually the focus shifts to the basics needed to build a sensor/actor network as described in the previous sections. This includes both the hardware part in Section 2.3 discussing the principles of microcontrollers, embedded systems, and communication. But what's the hardware without the software? Therefore in Section 2.4 Linux - and connected to that the concepts of both operating systems and free and open source software - will be introduced, followed by a short discussion on modern communication protocols.

Following the theory, Chapter 3 shifts the focus towards the more practical aspects of the problem to solve. This starts with a more detailed description of the problem itself in Section 3.1. Next the system requirements are laid out in detail in Section 3.2. Basically this is a translation of the human cognitive features into technical requirements, but will not spare a rough overview of existing Jumpcube components on both hardware and software side as they will need to be integrated with the system.

Chapter 4 is all about the results. Section 4.1 describes the implemented communication backbone with all its features and some benchmarks aim to check if the system requirements are met. After that the single units, referred to as the environment controllers, are presented in section 4.2. In this section one can also see how new controllers may be derived from the existing ones by small adaptations to their hardware or software components.

Closing this chapter is a brief overview on the control terminal in Section 4.3. This section not only aims to show the control terminal characteristics and architecture, but also tries to be a manual for both users of the system and engineers who want to further extend it or adapt it to their own needs.

The second to last Chapter 5 contains user feedback gathered through questionnaires on the Jumpcube's public appearances and in respect to that a short discussion on how the system influences the user experience by augmenting it with multi-sensory stimuli.

To wrap up the closing Chapter 6 shows what conclusions can be drawn and proposes some further work for extending the multi-sensory experience even more.

CHAPTER 2

Literature review

2.1 Virtual Reality

2.1.1 What is VR?

Virtual Reality, or VR for short, is not a single technology, but rather a whole family of devices which all aim to simulate a real-life-like experience. Yet for this thesis we shall put our focus on head mounted displays, or HMD for short, since it is the design adopted by all major manufacturers targeting consumers today. Many [Bro] well-respected [RT] technology [Sch] and psychology journals [SK] trace their origin back as far as the late 1950s and early 1960's to the work of Morton Heilig and his "Stereoscopic-television apparatus for individual use"[Heib]. As can be seen in Figure 2.1 his device design already resembles modern VR headsets in their exterior appearance. And also the internal workings of the device as can be seen in Figure 2.2 are the same as in modern VR HMDs with the exception of Heilig's design containing a "television tube", which at the time was state of the art, as the Thin Film Transistor Liquid Crystal based displays have not been invented until 1988 [Kaw02]. But the rest of the components, namely (with there numbering in Figure 2.1 and Figure 2.2):

- optical lenses (21, 22) to workaround the fact that the image is very close to the user's eye
- handles for lateral (37) adjustments - the rest of the adjustments described by Heilig are not available on most modern VR HMDs
- a strap (16) to allow hands-free operation
- a connection cable (53)
- a cavity to leave room for the user's nose (13)
- an image producing unit (24)

remain unchanged up to date as can be seen in Figure 2.3. Therefore Heilig's design - if used correctly - would have been perfectly capable of creating a perception of depth

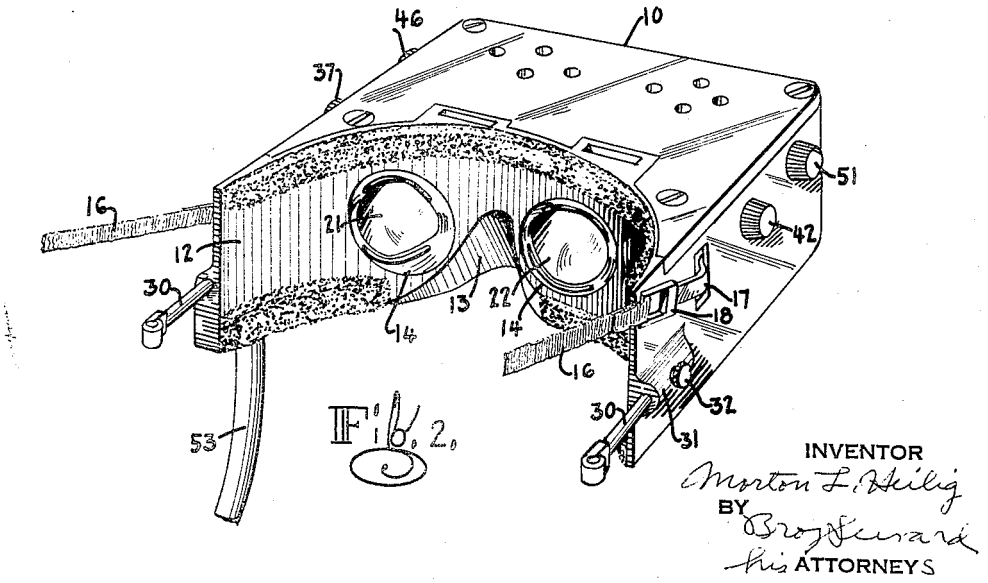


Figure 2.1: Morton Heilig, Stereoscopic-television apparatus for individual use, 1960, United States patent 2,955,156 - page 1.
[Heib]

within the displayed image. The reason for this is founded in the way how human visual perception works and what enables a human to visually perceive depth. This aspect is explained in more detail in Section 2.1.2.

For completeness the reader shall note, that Heilig's patent lacks one very important feature of today's VR HMDs, namely the ability to determine and propagate the direction the user is looking at. This comes as no surprise as computers, which are the things we use as image source for today's VR HMDs were something completely different then. The notion of an integrated semiconductor circuit has only been filed for patent in 1959 [Inc]. So it is easy to assume that even throughout the 1960s an image source which would create content based on the user's view direction was hard to imagine. Nevertheless, Heilig was able to build a complete multi-sensory experience around his 1957 invention a few years later in 1960. The Sensorama depicted in Figure 2.4 combined "the effects of the breeze, the odor, the visual images and binaural sound"[Heia]. The interested reader can find more details on multi-sensory Virtual Reality in Section 2.1.4.

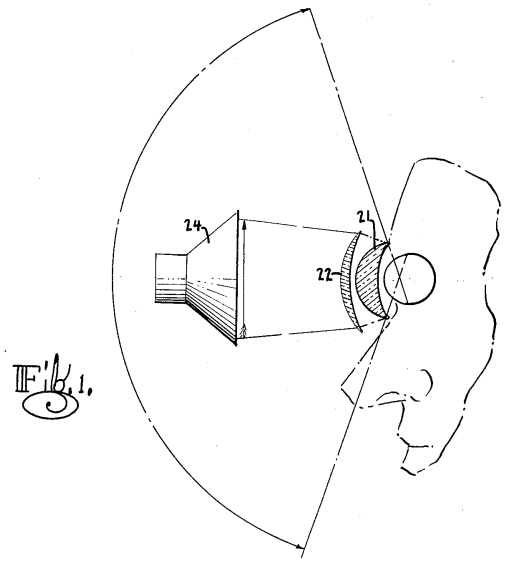


Figure 2.2: Morton Heilig, Stereoscopic-television apparatus for individual use, 1960, United States patent 2,955,156 -page 1.
[Heib]

2.1.2 Depth perception in the human visual apparatus

The human visual apparatus uses many features from the image perceived by the eye to extract spatial information. These features can be classified into two classes [HR95], namely:

- monocular cues rely on features of an image perceived by one eye
- binocular cues rely on differences between the images perceived by both eyes

Since monocular cues are intrinsic to the image, they are not enhanced by providing two dedicated image sources, i.e. one per eye, as presented in Section 2.1.1. Therefore they are out of scope for this thesis and we shall focus on the binocular cues and show their relation to Virtual Reality.

The first binocular cue that will be discussed is the so-called *stereopsis* [HR95], or retinal (binocular) disparity, or binocular parallax. The effect originates from the eyes being at different positions in space, which means that the image of their three dimensional surroundings is projected on each of them at a slightly different angle. Assuming we have a solid object, e.g. a cube, in an otherwise empty room. By laws of geometry the projection of the cube on the individual eye depends on the relative position of the eye and the cube. Since the eyes are not at the same position in space we can conclude that the images projected onto them will be slightly different. An illustration can be found in Figure 2.5. We can also deduce that the amount of difference between the images for any given object depends on the distance of the object from the eyes, or in other words: the

Figure 2.3: The HTC Vive uses lenses (bottom row, center) to project the image creating device further away from the user's eye. The housings of these lenses are mounted onto a mechanic which allows to move their position on the left-and-right axis. This enables the device to adapt to different eye spacings. Both approaches were described in Heilig's patent in 1957.
[iFi]

farther the object is from the eyes, the less the two images perceived by the eyes will differ from each other. An illustration of this effect can be found in 2.6. Charles Wheatstone was able to show in 1838 that the human brain is able to notice these differences and deduce the objects distance from them [Whe].

Proceeding to the next binocular cue we look at *convergence* [HR95]. For the ability to use the images from both eyes the human visual apparatus has to make sure some preconditions are met:

- the lens within each eye has to be adjusted in such a way that it produces a sharp image of mentioned object on the retina. The adjustment depends on the distance of the object to focus from the eye. This effect is known as *accommodation*.
- both eyeballs have to be rotated around their vertical axis to make sure the image of the object to focus is projected on the area of the retina with sharpest vision. This effect is known as convergence.

Figure 2.4: The Sensorama
[Min]

Even though accommodation and convergence interact in a closely coupled feedback loop to form the accommodation-convergence reflex [FW57], accommodation alone is a monocular cue since it relies on the information of each eye, i.e. its lens adjustment, individually. Convergence on the other hand relies on the angle between both eyes' direction of view, i.e. the rotation around their vertical axis and is therefore categorized as a binocular cue. The combined effect though leads to accommodation and convergence not being controllable on their own, but only in conjunction. This leads to the effect depicted in Figure 2.7: The closer an object is to the eyes, the further they have to be rotated towards each other to have the object projected onto the right spot. The brain is able to evaluate the angle between the eyes and from that deduct the distance of the object from the eyes.

Now, while Stereopsis can be leveraged quite easily by Virtual Reality Headsets following Heilig's design, as they have one dedicated image source per eye, exploiting convergence is more difficult. The reason for this is founded in how Head Mounted Displays are built. The image producing device is at a fixed position relative to the viewer's eyes. In addition the image producing device is at a very short distance from the eye. Heilig put lenses between the eye and the image producing device to virtually push the image producing device further away from the eye - a technique employed up to date as can be seen in Figure 2.3. Yet both the distance of the image producing device from the eye

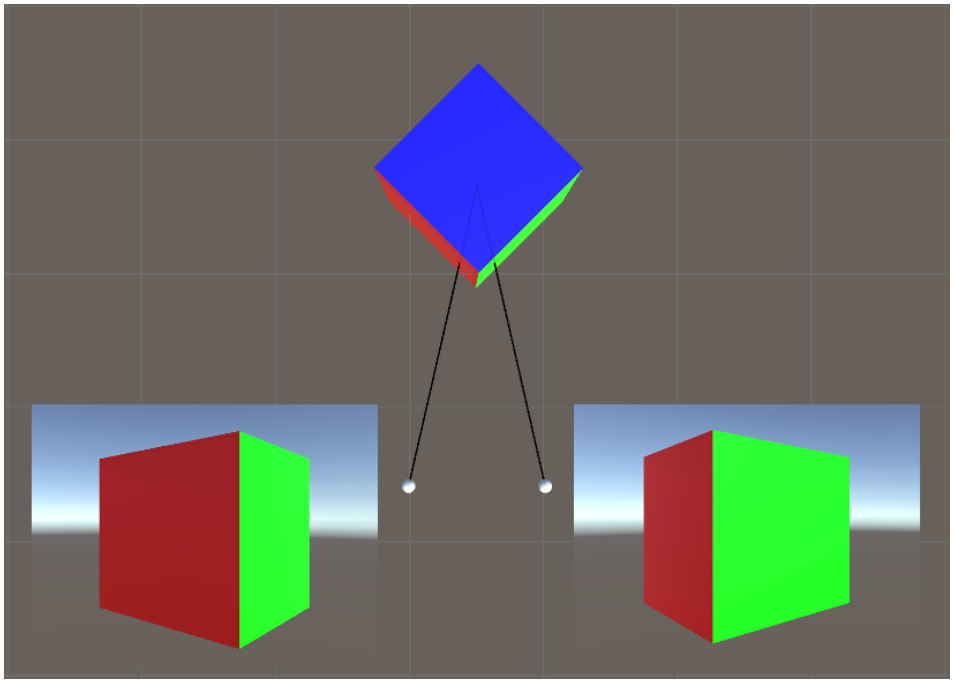


Figure 2.5: Due to the eyes (white) being at different positions in space the projections of 3D objects onto them differ. This effect is known as Stereopsis.

and the optical characteristics of the lenses are fixed, which in logical consequence leads to the whole image being on the same focal plane. This leads to a fixed accommodation of the viewers eye, which due to their coupled nature leads to fixed convergence of the eyes. The result is the so-called *vergence–accommodation conflict*, a well known effect in human visual perception[Hua17].

2.1.3 Development of Virtual Reality solutions

As we have seen in Section 2.1.1 Virtual Reality is a long known technology. Modern Virtual Reality Head Mounted Display do not differ much from the design in Heilig's patents in the way how they function and how they are built. Virtual Reality systems have been evolving throughout this over 50 year timespan, but none of these stages reached mainstream availability. As an example we may have a look at the Sega VR project by console manufacturer Sega. In 1991 [Hor] Sega announced the Sega VR project, a Virtual Reality Headset featuring dual LCD screens for 3D vision and stereo headphones for 3D audio. The device also featured an inertial sensor which should give the headset the ability to track its orientation and position. The system was scheduled to hit market in 1994, yet the project was abandoned after its last appearance at Consumer Electronics Show 1993. Sega stated the project was aborted due to it being too realistic and Sega fearing users could get hurt by running into walls while wearing the Virtual Reality headset. In 1995 Nintendo released its Virtual Reality solution, the Virtual Boy

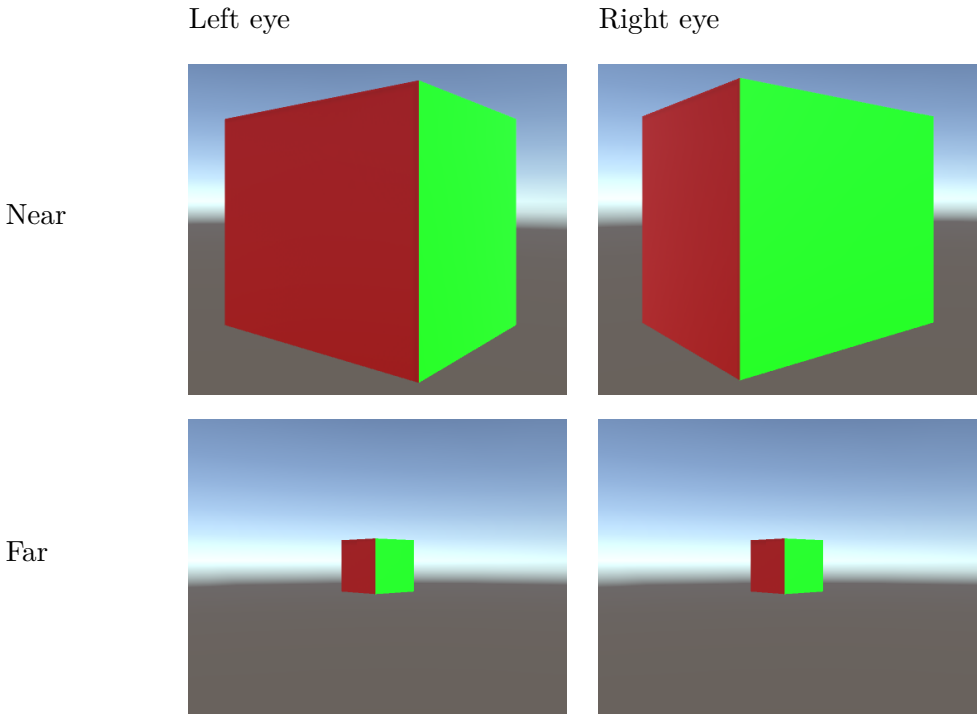


Figure 2.6: The further away an object is from the eyes the less its projections onto the individual eye will differ. From this the visual apparatus is able to deduce how far away an object is

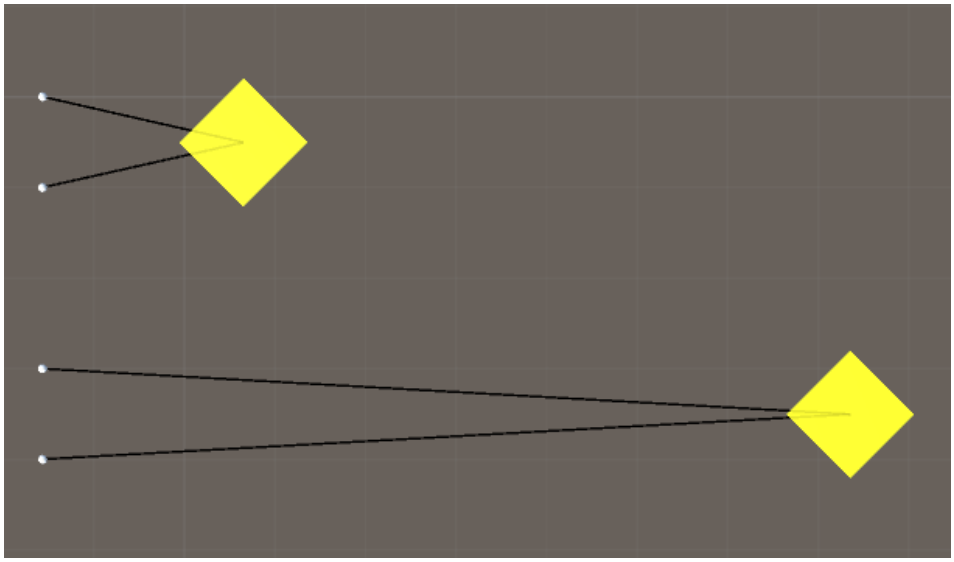


Figure 2.7: The closer an object is, the more the eyes need to rotate towards each other. This effect is known as convergence.

[Edw]. Different from the Sega VR the Virtual Boy was not a head mounted display, but a table-top device. Hence the user was not able to look or move around in the virtual world. With that the system shares many features with Heilig's Sensorama and is very different from today's virtual reality head mounted displays. The system used two scan line array monochrome displays, one per eye. Nintendo anticipated 3 million units sold in Japan within the first year [Nin]. As it became clear that the Virtual Boy will not meet the economical expectations Nintendo stopped all investments only six moths after the release. By the projects end in late 1996 only 770,000 units were sold worldwide [Edw].

Yet the interest in the technology did not die together with these project and at least scientists welcome the notion of it being "too realistic" for a variety of use cases. In 1995 Rothbaum et. al. showed how Virtual Reality can be effective in treating acrophobia, the fear of heights [RHK+95]. Soon after in 1997 and 1999 scientists where able to show a therapeutic benefit when using Virtual Reality on veterans of the Iraq war [ARS97] and the Vietnam war [BORB99] as a treatment for Post Traumatic Stress Disorder. The idea behind all of these projects was to analyze if and how it would benefit patients suffering from phobias or traumas due to their war zone experiences to experience the cause of their phobia or trauma in a controlled and safe, yet realistic environment. To achieve these therapeutic benefits all projects included the development of dedicated software able to simulate the desired scenario, i.e. a 50 story building and a war zone respectively. This software was designed to run on consumer grade hardware. For the Vietnam related project the scientists went as far as to build a custom piece of hardware they called "Thunder Seat", a chair which features a woofer speaker simulating helicopter vibrations to address the patient's senses beyond visual and auditive. The reason for this decision is highly relevant to the Jumpcube project and is discussed in more detail in Section 2.1.4.

To sum up, we saw that Virtual Reality hardware designs have been known since the late 1950s and by the end of the 1990s solid scientific evidence existed which showed that Virtual Reality felt "real" enough even to the extend of treating psychological disorders on consumer computers available at the time. Yet it took another roughly 20 years until a project to bring Virtual Reality to the mainstream gained interest comparable to the Sega VR. Virtual Reality headsets have been developed in the 1990s and 2000s as a list available on *stereo3d.com* shows [Bun], yet the cost for the devices which offer field of view and resolution comparable to the devices used in above mentioned studies shows their target is not the mainstream consumer. As an example it shall be noted, that the Kaiser Electro-Optics Inc. ProViewXL 35/50 from 1998, which offers similar specs as the Virtual Research Systems V6 used in the above mentioned Vietnam veterans Post Traumatic Stress Disorder project, was selling for 15.000$, a price well beyond the mainstream market. On the other hand the more affordable models on said list all feature only half the field of view, so the findings from above mentioned studies about the perceived reality might not be applicable.

In June 2012 at *E3* John Carmack, founder of *id Software*, showed a prototype of a Virtual Reality head mounted display he got from Palmer Luckey. The media echo was massive and in June 2012 the latter founded Oculus VR [Kum]. Luckey's prototype

took advantage of Nokia's, Apple's and Google's introduction of smartphones, i.e. the N-Series, the iPhone and Android based devices respectively, to mass market in the early to mid 2000s. These devices incorporate many features needed for a Virtual Reality head mounted display and the parts, upfront the displays, were suddenly available in a variety of sizes due to the differently sized smartphones and tablets, in large numbers and at reasonable price. This factor which has been heavily exploited when building the Oculus Rift Development Kit 2 in 2014, which contains the entire front panel of a Samsung Galaxy Note 3 including the Samsung branding on the display frame and the gaps which would hold speaker, front camera and distance sensor in the smartphone. By March 2013 Oculus VR had received the support of both Unreal Engine [Eng] and Unity [VRa], two of the leading game engines currently on the market.

2.1.4 Immersion and presence

As already mentioned in Section 1.3 we stick to the terminology proposed by Mel Slater [Sla03]. Slater tries to avoid confusion in discussions because of a too wide usage of either of these terms. He claims that a clear terminology would benefit discussions by avoiding that the participants talk about different things using the same term, hence ending up in what seems to be disagreement over a topic, while in reality each participant of the discussion referenced a different topic. So Slater proposes to use the terms *immersion* and *presence* as well-defined terms having no overlap in their meaning. To achieve this he proposes to let:

- *immersion* be the objectively measurable and quantifiable properties of a Virtual Reality system or component, e.g. a Virtual Reality head mounted display. Hence, immersion is intrinsic to the system itself and therefore does not change for different users.
- *presence* be the subjective experience of the user. Presence might still be quantifiable and measurable, e.g. physical reactions of a phobia patient when presented with a virtual equivalent of his/her phobia's trigger, but it is at all times subjective to the user and hence varies from user to user.

Bowman and McMahan [BM07] divided immersion into 9 metrics. Former Valve employee Micheal Abrash gave a talk on Steam Dev Days 2014 [Abr14], where he claims Valve's VR research team identified the values for some of these metrics necessary to establish presence:

Metric	**Description [BM07]**	**Value [Abr14]**	**Note**
Field of view	The size of the visual field that can be viewed instantaneously	$\geq 80°$	-
Field of regard Display size	The total size of the visual field surrounding the user -	- -	Up to 360° with head mounted displays -
Display resolution	-	$\geq 1080p$	-
Stereoscopy	The display of different images to each eye to provide an additional depth cue	-	Possible with head mounted displays
Head-based rendering	The display of images based on the physical position and orientation of the user's head	Millimeter and $\frac{1}{4}°$ accurate	-
Realism of lighting	-	-	Software dependent
Refresh rate	-	$> 60Hz$, $\geq 95Hz$	-
Frame rate	-	-	-

Abrash goes on to mention other metrics not present in the list of Bowman and McMahan crucial for establishing presence:

Metric	**Description**	**Value**	**Note**
Low pixel persistence	Time a pixel needs to become dark after being lit	$< 3ms$	-
Global display	All pixels are illuminated simultaneously	Present	Rolling display may work with eye tracking
Optical calibration	A highly accurate process for characterizing the lenses and correcting the rendered image	Present	-
Latency	Time between motion and last photon	$\sim 20ms$	$25ms$ may be enough

We conclude from above table that immersion can be easily quantized by just looking at the specification sheet of the Virtual Reality system and the software at hand. The only exception here is frame rate, though even this aspect can easily be determined on the running system. We have also seen that minimum requirements to immersion can be established for allowing presence, but what about measuring presence itself? As becomes clear from Slater's terminology presence is subjective and therefore intrinsically hard to measure and quantize objectively. Also Bowman and McMahan stated that they initially "were not sure how to evaluate that claim", being: presence helps people solve some tasks more efficiently [BM07].

Three different approaches have been presented to solve this problem with different advantages and disadvantages:

- self-assessment questionnaires [WS98] are easy to do, but the result is still highly subjective to the user.
- task performance based approaches [SLUK99] are more complicated to perform, since such a task has to be defined, but the result is more objective then a simple questionnaire. Yet this approach fails on Virtual Reality applications which are not designed to solve tasks, like Virtual Reality games or other applications designed purely for entertainment.
- physiology based approaches [MIWB02] like the measurement of heart rate or skin conductance is easy to perform and offers objective values. This approach though requires additional hardware able to measure the chosen physiological indicators. In addition some indicators are specific to a certain type of presence, like heart rate is to stress level.

Due to the Jumpcube's focus on entertainment the task based evaluation approach is not feasible. The self-assessment questionnaires and physiology based approaches can be used.

All of the examples we have seen so far focus on defining the immersion of visual aspects, yet the "Thunder Seat" from the Vietnam veterans Post Traumatic Stress Disorder project shows, that there has been scientific work on targeting senses beyond visual and auditive as well. In this context we shall have a look at the work of Swedish scientist Henrik Ehrsson. Ehrsson and his group at Brain, Body and Self Laboratory in Stockholm work on generating out of body experiences on users by building upon the rubber hand illusion. The rubber hand illusion was discovered in 1998 by Botvinick and Cohen [BC98]. They discovered a person can be tricked into believing that a rubber hand is part of his/her own body with relative simple methods: They put a rubber hand next to the test subject and hid his/her actual hand from his/her view. Then they applied contemporaneous stimuli to both the rubber hand and the actual hand hidden from the person's sight. The subjects started experiencing a sensation as if they felt the viewed stimulus, which is the one applied to the rubber hand, and not the one applied to their actual hand. Put differently, the subjects started projecting a part of their body onto a piece of rubber.

Ehrsson found he could apply the rubber hand illusion not only to limbs, but to the whole body [Ehr07] and that empty space can be used as a projection target as well, as illustrated in Figure 2.8. Ehrsson also found the synchronicity of both stimuli, i.e. the one on the actual body and the one on the projection target, is crucial for the effect. Key to all of these findings is, that the subject knows at any time how the experiment works and that the projection target is not actually part of the subject's body. Nonetheless, the mind can be tricked as far as to trigger both physiological and self-defensory reactions if the projection target is "hurt" [Ehr07].

Watanabe and Tachi [WT11] during their work on telepresence using robots also found the perceived notion of presence varies depending on the type of tactile feedback delivered to the subject. They used two robotic arms where one is able to mirror the other's movement. The subject sits between the two robot arms in such way that one points at the subject's back and the other is in front of the subject pointing away from him/her. With this setup the subject is able to poke his/her own back. The setup is depicted in Figure 2.9. This allowed them to vary the type of tactile feedback the subject experiences, i.e. by enabling or disabling the robot's motion mirroring feature and by placing or removing a physical object in front of the robot arm operated by the subject to simulate resistance when hitting the back. They showed that both the subject's perceived location, i.e between the robot arms or behind the rear robot arm, and the subject's perceived body count, i.e. the approval of the statement "I felt that I had two bodies", depends on the types of tactile feedback present.

To sum up we have seen that visual sensory information is crucial to the amount of presence delivered by a Virtual Reality system and goes as far as to be usable in therapy of phobias or even psychological disorders. Yet the focus on visual sensory information is too restricted when trying to generate a full out-of-body experience using a Virtual Reality system and hence tactile feedback and maybe other sensory information as well can contribute to the amount of presence experienced by the user when being put in a virtual environment.

2.1.5 Simulator sickness

Simulator sickness is a special kind of motion sickness induced by the use of a simulator. Different kinds of motions sicknesses, like seasickness which the ancient Greek Hippocrates (460 - 377 BC) described as "Sailing on the sea proves that motion disorders the body" [LD05]. Other well known types of motion sickness are car sickness, or airsickness. While the term *sickness* usually refers to a condition of disease or malady, this is not the case for motion sickness as it affects otherwise perfectly healthy humans [LD05]. The only requirement known to date is an intact vestibular system, as many experiments failed to induce subjects suffering from labyrinthine defects to experience it [KGMB68]. Motion sickness is unpleasant to the subject as its symptoms range from nausea and cold sweat up to headache and vomit, the latter being rare for simulator sickness [LD05]. While its true origin still remains to be proven, the widely accepted hypothesis for the cause of

Figure 2.8: By exploiting the rubber hand illusion a subject can be tricked into projecting his/her body into empty space.
[BaKI]

motion sickness is a mismatch on different sensory information, like e.g. motion being reported by the eye, but the vestibular system senses no motion [Ben02].

Treisman proposed an evolutionary explanation as the origin of motion sickness. He hypothesizes the mechanism in the brain responsible of motion and orientation is the same as the one dealing with dysfunction caused by the ingestion of toxic substances. He goes on to define motion sickness and its associated symptoms as protective actions against poisoning [Tre77]. An alternative theory has been proposed by Ebenholtz et. al. in 1997 which postulates motion sickness as being a consequence of tight coupling between vestibular system and involuntary ocular movement in place to gain visual stability during movement [EMCL94]. Regardless of the cause, motion sickness can be treated by adaption, a treatment used both voluntarily and involuntarily over centuries, e.g. by sailors. The author can confirm from experience adaption also reliefs the motion sickness symptoms evoked by Virtual Reality environments. In addition there are drugs effectively relieving the subject from the symptoms of motion sickness, though it shall be noted, that e.g. alcohol has a negative influence on subjects [LD05].

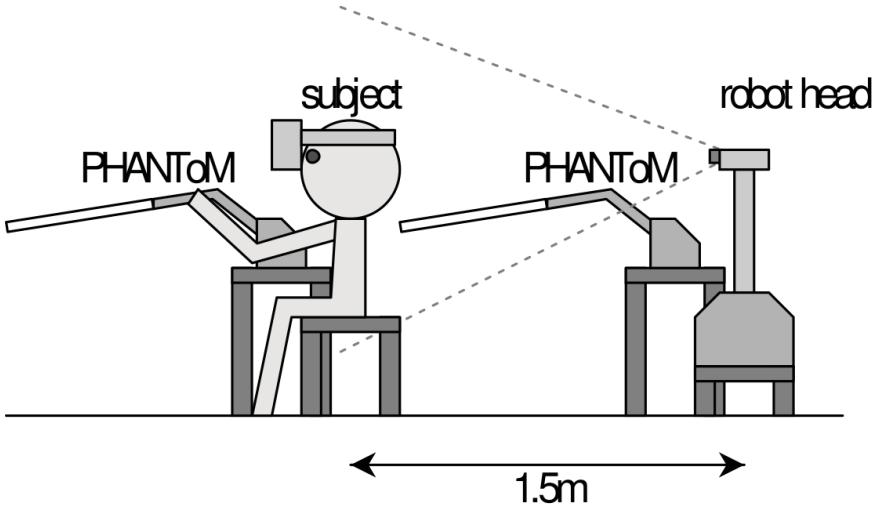


Figure 2.9: Schematic of the setup used by Watanabe and Tachi. The subject is wearing a Virtual Reality headset displaying the image recorded by a two lens camera system located at the "robot head". The devices referred to as PHANToM are robot arms able to mirror each other's movements.
[WT11]

Motion sickness plays an important role in Virtual Reality environments, since it obviously affects presence perceived by the user. It has been shown that inducing vection (i.e. the feeling of self-movement caused by the entire surrounding moving) can cause motion sickness [BBP+08], yet other causes are less clear, as research was done on the visual complexity of animated actors inside the virtual environment, which contradicts previous work it was trying to build upon [KKC09]. Similarly, scientific work exists which claims to have found a correlation between the system's latency and the motion sickness induced by it [Ell09], while other studies were not able to reproduce those findings, postulating no connection between latency and motion sickness [MRWB03]. Hence it is difficult to tell what actually causes simulator sickness in Virtual Reality environments. Nonetheless, head mounted display manufacturer Oculus VR attributed the reduction in motion sickness from their DK1 to their DK2 model to the adoption of a "Low Persistence OLED Display" [VRb].

This concludes our overview on Virtual Reality. We have seen how the hardware evolved starting from the late 1950s to its current state and we defined immersion and presence as metrics for the user experience in virtual environments. We have also discussed how physiological effects like motion sickness need to be addressed and how psychological effects like the rubber hand illusion can be exploited when developing a Virtual Reality application. We shall now direct our focus to Game Engines, how they became to be the

backbone of interactive 3D content development, and why they are relevant for Virtual Reality in regard to the Jumpcube project.

2.2 Game Engines

2.2.1 Definition

In the early days of game development up to the 1980s games were often written from scratch in assembler language without any aid from other software packages.[BEW+98]. This was due to the fact that games targeted very specific hardware platforms like e.g. the Atari 2600 or even custom build hardware like arcade hardware. Computing resources, especially memory, were a very expensive commodity and therefor scarce by definition. Hence games had to be highly optimized to the specific platform they targeted [Gre14]. The distribution media for home computer games at the time were cartridges packed with Read Only Memory, which meant games could not be updated or patched in the field. These two factors meant that with the release of a game all of its software could be put in an archive never to be touched again.

The situation began to change in the 1980s with the arrival of 2D game creation systems like the War Game Construction Kit released by ASCII in 1983 [Mus]. These kits provided a framework to build games. Even though they usually restricted the developer to a very specific genre like the Pinball Construction Set [Unk82] they provided a reusable framework which allowed hobbyists to create their own games.

The term "game engine" first appeared in the mid 1990s with the rise in popularity of first-person shooters like Doom by id Software [Gre14]. The developers of Doom divided the software into two parts, one being the core components like rendering, collision detection and the audio system, and the other being more game specific parts, like 3D models and sound assets, the game's world and its rules. This allowed for the development of games by just switching out the latter and make some small changes on the first if needed instead of having to build the entire game from scratch. This separation also allowed the company behind the game engine to license mentioned core components to other companies, which became a significant economic factor for some game engine developers [Gre14]. On the other hand game engines do provide a benefit to the licensee as the game engine abstracts software and hardware capabilities of the target and can adapt to the underlying hardware platform and software APIs provided by the respective operating system. Hence a game can be developed in a platform independent process and then be compiled for various platforms. In addition, a game engine might provide the ability to adapt to the hardware capabilities of the host system by giving the opportunity to reduce resource consumption by e.g. lowering the resolution the game is rendered in. As of the 2017.4 release the Unity engine for example supports 18 different build targets including all major desktop and mobile operating systems as well as all major gaming consoles [Unib].

A game engine is usually not a monolithic piece of software, but consist of a variety of modules. The top level division is between the tools used by the game's developers which will be referred to as the integrated development environment (IDE) in the context of this thesis and the so-called runtime, which is a set of software libraries providing the before-mentioned abstraction of the underlying hard- and software [MSK15]. The IDE usually consists of an editor for the so-called scene graph which allows to build worlds, to create objects therein, and put them into a relation to each other [BEW+98]. The IDE also provides a scripting API with which a developer can create dynamic interactions or reactions to user input [MSK15]. This API offers data structures and algorithms optimized for the use in computer graphics and interfaces to third party libraries e.g. for physics simulation or special hardware support. The IDE also allows to change at least some of the inner workings of the game engine's runtime by providing e.g. custom graphic shaders. From the IDE a game can then be compiled for a specific or more target platforms. In this step the assets, the scripts, the scene graphs, and all other custom created parts of the game are packed together and tied to the runtime. This combination then yields a runnable binary to be distributed over the desired distribution channel.

The runtime of a modern game engine is a fairly complex piece of software providing a variety of features. Following Gregory's taxonomy [Gre14] we can distinguish low level renderer, audio subsystem (in 3D enabled engines usually with spacial audio support), visual effects like particle systems for smoke or fire, a front end for 2D user interfaces such as menus or head up displays, collision and physics, animation, human interface device handling, a network subsystem for online multiplayer support just to mention a few. Each of these can be further split up into more atomic subparts.

2.2.2 Usage motivation

Having such powerful software systems as game engines at disposal is a huge advantage in development. The game developer does not need to create a complete rendering pipeline to get state of the art photo-realistic computer graphics with very little effort, assuming both graphical and sound assets already exist or are sourced from a third party. This fully removes the need for a dedicated software development department and leaves only the need for an art department, which is needed anyway if custom assets are to be created. As an example we shall look at an architectural office, which would like the customer to be able to walk around a building in its final state which currently only exists as a CAD plan. Assuming the CAD plans exist as a 3D model, which is fair to assume, the assets already exists. All the rest, like rendering of different surfaces (think wooden floor and stone walls), lighting conditions at different times of the day or even different seasons of the year), the ability to move the observer through the scene by using some human interface device, and putting different kinds of furniture in the building can be easily achieved with the use of the right game engine. A result of such an endeavor can be seen in Figure 2.10 representing a screenshot of the Unreal Paris 2018 project by Benoit Dereau.

As a further advantage the use of a game engine enables any project to easily be adapted to different hardware and software platforms, even those not available at the time the project started. Also, with the appearance of a new technology it can easily be integrated with the project without the need to change the project itself. On the Unity engine for example one can add Virtual Reality support for all major consumer products, being the Oculus Rift series, the HTC Vive series, and the Playstation VR platform, to an existing project by simply enabling Virtual Reality support in the build settings of the project and recompile the project.

Finally, the project creator can also take advantage of the continuous evolution of the game engine used. With the advancement of hardware capabilities new possibilities arise which were previously not possible in real time computer graphics. Existing technologies reserved for enterprise grade hardware slowly trickle down into the consumer sector, allowing them to be adopted by a much wider range of users. As an example for this evolution it shall be noticed that graphics processor manufacturer NVIDIA announced a new technology package called NVIDIA RTX for GDC 2018 [Bur18]. NVIDIA RTX aims to bring ray tracing in real time to consumer grade hardware with NVIDIA's Volta graphics processor architecture. Unity and Epic Games, manufacturer of the Unreal Engine, already announced their support for NVIDIA RTX, meaning that the technology will be available to all projects based on these engines [Bur18]. Of course the introduction of such a new technology stack or new hardware like virtual reality headsets means that at least parts of the rendering pipeline will have to be adapted for the feature to be usable within the project. Yet, since the rendering pipeline is part of the game engine, this adaption is made available to all clients by the engine's manufacturer. So in the best case scenario the project creator will only need to update the engine's IDE and recompile the project to take advantage of the new technology.

2.2.3 Description of examples

We introduce Unity, Unreal Engine 4 and CryEngine V, because they are typical modern multi-purpose game engines featuring support for Virtual Reality. They were chosen because they were subjectively brought to our attention the most in the recent past. The reader shall note that this is by no means an extensive list, as Zarrad [Zar18] identified 20 different modern game engines available on the market, which meet all the posed requirements, being feature complete, having been used for "high-quality games", and being actively developed.

First in line is **Unity** by Unity Technologies. Unity focuses heavily on ease of use and platform compatibility. Unity offers a rich tool set allowing to optimize the project for each of the target platforms. As of version 2017.4 Unity offers scripting support in C# and JavaScript. Sooner versions also supported Boo, a type-safe Python dialect as reported by Gregory in [Gre14], but it has been deprecated in version 5.0. JavaScript support has also been on the way of becoming deprecated since Unity 2017.1 was released in August 2017. Unity over the time also dropped one of their unique points, namely the Web Player, a browser plugin enabling to run Unity binaries from websites, similar

Figure 2.10: The Unreal Paris 2018 demo by Benoit Dereau. This demo was built using the Unreal Engine 4. The image is computer-generated in real time on consumer grade hardware.

[Der]

to Flash or Java Applets, in version 5.4 [unia]. Even without the support for Boo and JavaScript, Unity's support for Mono, being an open source implementation of Microsoft's .NET Framework, enables the developer to use many libraries from the .NET ecosystem. It shall be noted, that Unity has a dedicated mode for creating 2D applications, but due to this thesis' focus on Virtual Reality only 3D capabilities of the engine are relevant.

The **Unreal Engine 4** by Epic Games is the latest evolution of an the Unreal Engine family, dating back as far as 1998 to the release of the name giving first-person shooter Unreal. Most developers modify the engine in various ways for running their game on a particular platform [Gre14]. Nonetheless Unreal Engine 4 is a very powerful game engine, and performs better then Unity in the domain of photo-realistic rendering [SBH+17]. Unreal Engine uses C++ as a scripting backend providing a custom API, but one can also choose the graphical node-based programming environment called Blueprint Visual Scripting. The C++ API also offers Blueprint specific markup, allowing both systems to integrate with each other [Gamb]. The Blueprint Visual Scripting allows designers without programming experience to create dynamic interactions using graphical programming, though from the author's experience the so-called Blueprints tend to explode in size and hence become confusing for more complex interactions. Featuring its own API, not many libraries from foreign ecosystems can be adopted in Unreal Engine 4. However, it offers an API for creating plugins.

The last candidate in this roundup is **CryEngine** by Crytek. The development of CryEngine started as a tech demo for graphics processing units manufacturer NVIDIA, but Crytek realized the potential and turned it into a full game called Far Cry [Gre14].

CryEngine V (spoken CryEngine five), the latest offspring of the CryEngine family, is a feature complete and very powerful game engine. The latest version also introduced native support for Virtual Reality following Unreal Engine and Unity, how as we saw in previous chapters already had support for it. The lack of Virtual Reality support caused CryEngine 3, which was the latest version back when the project started, to not be considered at the time. CryEngine, similar to Unreal Engine, is said to have advantages over Unity in the domain of photo-realistic rendering [SBH+17]. CryEngine V supports C++, C# and Lua as scripting languages. As a second drawback CryEngine V does offer a plugin system, yet at the time of writing it is still flagged as beta feature and explicitly not recommended for production use [Cry]. This fact may render it very difficult to integrate the engine with custom hardware.

After discussing what game engines are, their history, and why they became crucial for modern interactive 3D content creation we shall now shift our focus towards the building blocks of custom hardware enabling us to built custom devices for the Jumpcube. These should allow us to address senses beyond visual and auditive. We shall first analyze the hardware parts available on the market, understand their inner workings and their respective advantages and disadvantages. We shall then move on to machine-to-machine communication from a hardware perspective followed by communication protocols, which represent the software perspective.

2.3 Controller hardware platforms

2.3.1 Microprocessor, microcontroller, and system-on-a-chip

Nearly 50 years ago, in 1971 Intel Cooperation took advantage of recent developments in computer architecture and integrated circuit manufacturing and released the 4004, the first commercially available "computer on a chip" or *microprocessor* for short [Aya04]. The Intel 4004 was a 4 bit digital general-purpose digital computer central processing unit (or CPU for short) and today is widely recognized as the first fully integrated microprocessor. Soon other semiconductor manufacturer followed Intel to release their own microprocessors, so that by the end of the 1970s different models were available [Aya04]. Ayala [Aya04] defines the minimum required set of components of such a microprocessor as follows:

- arithmetic and logic unit
- program counter
- stack pointer
- working registers
- clock timing circuit
- interrupt circuits

From the list above we see that a microprocessor is by itself not a complete computer. The Intel 4004 for example had to be paired with memory, storage and shift register chips to form the MCS-4 microcomputer system [NH81].

As an offspring of the proceeding integration in 1974 Texas Instruments launched the first *microcontroller*, the TMS1000. The TMS1000 incorporated all of CPU, random access memory, read only memory and I/O ports [Des05]. With the integration of memory and I/O Microcontrollers form true computers-on-a-chip. Although microprocessors (paired with memory, storage and I/O) and microcontrollers are equivalent from a theoretical computer science standpoint, microprocessors only have very few instructions for bit handling, but feature many instructions for moving data from and to memory [Aya04] [God08]. Microprocessors are the opposite, namely they only have one or two instructions to move data from/to memory, but a variety of bit handling instructions. Ayala [Aya04] (page 6) summarized this as *"the microprocessor is concerned with rapid movement of code and data from external addresses to the chip; the microcontroller is concerned with rapid movement of bits within the chip"*. In addition Godse et. al. [God08] found microcontrollers to be less flexible from a design point of view, yet they require less additional hardware and hence need less space on the circuit board and have increased reliability. Hence, microprocessors became the choice for general purpose computers like PCs, while microcontrollers are used as single purpose devices in home appliances, computer peripherals such as printers or automobile engines [Uda09].

A slightly different approach came into life in the year 2000 as Cirrus Logic Inc. patented the *system-on-a-chip* [Kla04] or SoC for short. The patent shows that SoCs share some features with microcontrollers, namely the integration of a CPU with I/O and memory on a single substrate. Yet differently from microcontrollers, SoCs may integrate a microprocessor (though it may be microcontroller too) and additional processors, like e.g. graphic processing units or networking. It is also not required that a SoC incorporates all necessary components into one single chip, as the patent explicitly mentions a boot ROM, but only requires an "external memory interface". This means the RAM itself may be placed on a dedicated chip.

Today a wide variety of microcontrollers exist from different manufacturers like Atmel, Texas Instruments, Microchip, and more. In addition british company Arm licenses their Cortex-M series to a variety of manufacturers. Arm, which focuses on the development and licensing of so-called IP cores, claims that their Cortex-M series "have already been shipped in tens of billions of devices" [Arm]. Though having just the bare chip brings the need for additional effort when trying to prototype any microcontroller based custom piece of hardware, since the designer first of all has to build a circuit board and circuitry for powering and communicating with the microcontroller. This problem was recognized by Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino, and David Mellis who in 2005 launched the Arduino project [Bar12]. Their goal was to provide an easy to use microcontroller platform for hardware prototype development, mainly focusing on the academic environment. As the team had no intention of ramping up production after their needs were fulfilled they released Arduino as open source [Sev14]. Today the Arduino is widely used because of the following facts [JL13]:

- being open source
- the community which formed around the project

- the libraries contributed by said community
- its low cost
- its out of the box features such as I/O pins, PWM, I2C and SPI

Microprocessors up to date are still available on their own. For most desktop and server PCs this brings the need for a motherboard bearing a set of supporting circuitry and chips to form a fully functional PC. Yet some microporcessors available today are not bare CPUs anymore. Intel for example moved the memory controller into the CPU in their 2010 Core-i lineup [Int10]. For their 2017 Zen architecture based processors Intel's competitor AMD went as far as to incorporate SATA and USB connectivity into their products [AMDb]. Technically a modern product like the Zen based AMD Ryzen 2700U, additionally containing a memory controller and a GPU as well [AMDa], is closer to being an SoCs rather then a classic microprocessor. A truly SoC based part highly relevant in the scope of the Jumpcube project is the Raspberry Pi. The Raspberry Pi is a credit card sized single board computer initially intended to be used in computer science introduction courses [RW12]. In addition to SoC and RAM it incorporates USB, wired Ethernet network, video and audio output and boots from an SD-Card [RW12].

Comparison

To compare these devices we shall have a look at their specifications and discuss their respective advantages.

Product	Raspberry Pi 3	Arduino Uno
Class	SoC with microprocessor	Microcontroller
Power	0.58A@5V (≈3W)	≈15mA@5V (≈0.075W)
I/O	Digital I/O, PWM, SPI, I2C, UART, audio out, video out, USB Host, Ethernet, WiFi, Bluetooth	Digital I/O, PWM, SPI, I2C, UART, Analog in, USB Client
RAM	1GB	2kB
ROM	None	32kB Flash + 1kB EEPROM
CPU	1.4 GHz quad-core	16MHz single core
Price	≈30€	≈35€

We can see, that the Raspberry Pi has advantages on computing resources for both CPU and RAM. Though one needs to keep in mind, that it needs an operating system to run. The Arduino on the other hand consumes very little power and features analog inputs. In addition the Arduino does not need an operating system. As Richardson [RW12] points out, the Arduino is the better choice for simple automation, like a thermostat. However, if the data shall be presented via a web interface the Raspberry Pi is the better choice. Both parts are similarly priced, though it shall be noted that no-name copies of the Arduino are available at lower cost.

2.3.2 Communication

After having explored what building blocks are available for custom hardware components we shall now turn our attention to the means allowing them to communicate with each other. Since the simulations created for the Jumpcube will be based on modern game engines we will have to consider not only the capabilities of the custom-built devices, but also those of the game engines. Also, in order not to artificially restrict the game engines to chose from the communication channel should be as widely supported as possible. The Jumpcube, being designed as a movable structure, needs to be disassembled and moved. This means the communication channel needs to be both easily dis- and reassembled, and adapt to different and a priori unknown environments and their conditions. Discussed below we see two technical approaches to implement such a communication network based on international standards. In addition both approaches have been widely used for some 30 years and even longer and have proved to be reliable.

Fieldbus

The term *fieldbus* was first coined at an IEC meeting in 1985 [Tho05]. At the time the industry was seeking to replace the star-like point-to-point topology predominant at the time for connecting process control computers to the sensors or actors. These sensors and actors are referred to as field devices [DS00], which led to the name. One of the main goals was to reduce the cabling necessary to connect all of these sensors and actors by using a single shared medium which all nodes on the network plug into [DS00]. Back then at the lowest level of communication many standards were established, like for example the 4 - 20 mA standard for analog sensors or the 0 - 24 V for digital inputs. Both of these needed two dedicated cables per connected device [Tho05]. This led to high costs in both installation and maintenance. In a discussion draft published in 1986 by the International Society of Automation (ISA) this challenge was identified as the most important one when it comes to defining requirements for fieldbusses. In total seven potential benefits of fieldbusses over existing topologies were established by gathering feedback from all ISA members using a questionnaire. The full list, ordered by importance from greatest to least was [Tho05]:

- lowering the installation costs
- ease of adding field devices
- providing two-way communication with field devices
- improving the accuracy of information delivered at control room
- enhancing the maintainability of field devices
- providing remote access to measurement data through handheld interface
- more advanced control strategies can be implemented because of improved field data

In Europe at the same time the International Electrotechnical Commission (IEC) also directed their attention towards fieldbusses. The IEC identified the need for two classes of fieldbusses, being H1 and H2 respectively. The H1 provides low data rates for the

connection of some sensors mainly for process control, while H2 is a high speed fieldbus for manufacturing or the interconnection of H1 networks [Tho05]. Even though they provide similar functionalities, H1 and H2 differ in the number of devices, distance between devices, speed, and services available to the user.

By the end of the early standardization process both ISA and IEC came to similar conclusions on what a fieldbus should be and what it should be able to do. And even while the initial standardization only regarded process control and discrete manufacturing as applications for fieldbusses, later applications such as building automation or in-vehicle communication system showed to have very similar requirements. Yet, fieldbusses at the time were not considered as real-time networks [Tho05].

A few years before the standardization of fieldbusses, in 1978 the International Organization for Standardization (ISO) started to work on what would become the Open System Interconnection model (OSI). The model aims to bring all concepts necessary to develop communication protocols [Tho05]. The ISO/OSI model separates communication into 7 layers, where each layer services the one above and is serviced by the layer below [Zim80]. The layers are:

1. Application Layer: Top level layer, which serves the end user.
2. Presentation Layer: Give meaning to the data exchanged.
3. Session Layer: Assist in the support of interactions between presentation entities.
4. Transport Layer: Transport data transparently between session entities.
5. Network Layer: Provide independence from routing and switching considerations.
6. Data Link Layer: Establish, maintain, and release data links.
7. Physical Layer: Transmit symbols over the physical medium.

From a logical point of view the entities on a given layer communicate with each other by means provided to them by the layer below. A device on the network may implement all or only a subset of these layers depending on its purpose. A repeater in its simplest form for example only needs to read incoming symbols from the physical medium on the receiving end and write them to the physical medium on the sending end. For this it does not need to understand any but the lowermost Physical layer. On the other hand, a routing device which needs access to the routing information within the transmitted data needs to have support for the lowermost three layers, being Pyhisical, Data Link and Network. See Figure 2.11 for a sketch of such a communication.

The early offspring of fieldbus standardization focused on solving the problems of their time. By the mid 1990s such fieldbusses usually interconnected only about 6 nodes on average [DS00]. Due to the restricted capabilities of these nodes and the developers not expecting a complex network structure, they usually only employed a subset of the layers defined in the ISO/OSI model, mostly the bottom two or three plus the Application layer [DS00]. Accordingly, Thomesse [Tho05] claims it is common to say that a fieldbus has three layers, being:

- the Physical layer
- the Data Link layer, including implicitly the medium access control (MAC) layer

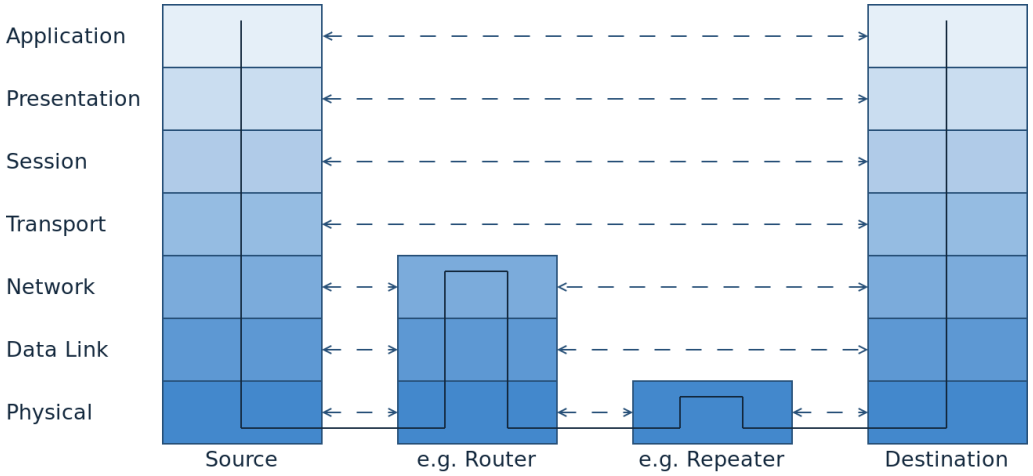


Figure 2.11: The ISO/OSI model. The dashed lines show the logical communication channels. The solid line shows the actual flow of data, where vertical lines show where data is processed, and horizontal lines show where data is transmitted.

- the Application layer

Yet, it soon turned out that the development costs for the nodes were not the driving factor, as maintenance of the node's software and the software tools to integrate and configure the network proved to become ever more expensive [DS00]. With this insight the concept of "plug & play" comes into play. The idea behind it is, that any new device can be attached to an existing network and the device would then integrate itself into the network without further user interaction. Though, as Dietrich et. al. [DS00] pointed out this is mere impossible to achieve, as any device will need at least some rudimentary information about the behavior expected from it.

Another challenge for fieldbusses arises from their very nature of using a single shared medium for communication, namely *collisions*. A collision happens if two or more devices write onto the bus at the same time. In such a scenario the state on the medium is a superposition of all written messages, in which case the individual messages cannot be reconstructed by the devices trying to read from the medium [DIL81]. To solve this issue three classes of approaches exist [Tho05]:

- *Carrier-sense multiple access* is an arbitration rule, where each node probes the medium (e.g. the fieldbus) prior to initiate transmission and only starts writing onto the bus as the medium is sensed idle.
- *Time-division multiple access* divides the access time into slots and assigns one (or more) of these slots to each device. A device is only allowed to write to the medium during its time slot.
- *Controlled access* means, that some deterministic mechanism other then time-division is in place to assign write privilege to a device. This can be achieved by passing around a single token and only the device holding the token is allowed

to write to the medium, or by having a single device (usually referred to as master) explicitly asking a device (in this scenario usually referred to as slave) for information.

Carrier-sense multiple access can be regarded as decentralized. Time-division multiple access can be regarded as centralized, as each node needs to adhere to one global slot schedule. Though the time source may be decentralized by each node having its own dedicated clock, or centralized by having a common clock source. Token-based Controlled access protocols can be regarded as decentralized, as the token is passed on either explicitly by the token holder after it finishes its activity on the medium, or implicitly by e.g. moving it according to the numerical order of the addresses of the devices on the bus. Polling-based Controlled access protocols on the other hand can be regarded as centralized as a single node manages write access to the medium [Tho05]. It shall be noted though, that different protocols can be combined. Profibus for example divides all nodes on the bus into masters and slaves. It uses a token-based protocol to elect the currently active master. That node then polls all nodes relevant to its operations [TV99].

Ethernet

In the 1970s the local area network (LAN) emerged as a solution for sharing expensive devices like printers and to overcome the increasing problem of wiring caused by the rising number of terminals in offices. Some ten years later three standards were established, being [PK09]:

- IEEE 802.3 Ethernet
- IEEE 802.4 Token Bus
- IEEE 802.5 Token Ring

For the scope of this thesis we shall restrict our focus on the IEEE 802.3 standard as it is today the dominant standard for LANs, even though in an evolved form compared to the original IEEE 802.3 standard [PK09]. In the beginning thick coaxial cables were chosen as a shared transmission media. On those the first Ethernet standard was able to transmit 10 Mb/s over up to 500 meters. In the mid 80s with the increasing popularity of LANs the manufacturers moved towards thinner and cheaper coaxial cables. The cost of these cables also coined the name *cheapernet*. This reduced not only the costs, but also the maximum length to 185 meters, while keeping the transmission rate at 10 Mb/s [PK09]. In the 1990s the industry moved away from the coaxial cable and switched to easier-to-use twisted-pair cables and a star-topology called hub-and-spoke LANs. Due to the new cables the maximum length was now restricted to 100 meters, still keeping the transmission rate at 10 Mb/s [PK09]. In the future years the physical layer was continuously improved leading to 10 Gb/s over UTP cables in 2006 with IEEE 802.3an [PK09].

On the data link layer Ethernet up to date still uses the same arbitration protocols for collision handling as in the 1980s in environments where such an arbitration is needed. This method is called carrier-sense multiple access/collision detection or CSMA/CD for

short [LMT01]. This means each node who wants to send a message first has to listen to the network. If it is busy the node waits until it becomes idle again. Then the message is sent. If, for example, two nodes are waiting, because a third node keeps the network busy, both start sending after the third one has finished. This will cause a collision. Therefore each node needs to listen for collisions on the network while it writes to it. If such a collision is detected the node aborts sending and then waits a random amount of time before starting from scratch [LMT01]. Also if a collision is detected the node sends out a long enough jam signal to assure all nodes on the network detect the collision. The range from which the random wait time is chosen is increased on every collision by a factor of two up to the tenth collision. Afterwards the range does not change. If a packet could not be sent after 16 tries the transmission is reported to the upper layers as failed [PK09]. When moving to the hub-and-spoke topology the collision detection was offloaded from the nodes to the hubs [PK09]. IEEE P802.3x in 1997 fully removed the need for collision management. This was accomplished both by implementing full-duplex and by replacing the hubs by so-called switches. Differently from a hub, which relays all incoming messages to all outgoing ports, a switch only relays the incoming message to the port to which the destination for said message is connected. If a new package for that destination arrives while the previous one is still being sent the second package is held back until sending is complete. Cables in such a network are point-to-point from device to switch or from switch to switch. This allows for full-duplex communication without collisions [Dec05].

We have now seen what communication hardware types are at our disposal. Since Ethernet only covers the bottom layers of the ISO/OSI model [PK09] we shall now look at protocols and software allowing us to complete the ISO/OSI model stack.

2.4 Controller software components

We can see from Flammini's work [FFS+02] that fieldbusses such as Profibus-DP and CAN2.0B offer a reduced implementation of the ISO/OSI model with four layers. Ethernet only covers the bottom two layers of the ISO/OSI model, which brings the need for implementations of the remaining five layers [FFS+02]. We shall now see what protocols are available. But first we will have a brief look at the operating system for the beforementioned Raspberry Pi.

2.4.1 Linux

The Raspberry Pi Foundation, manufacturer of the Raspberry Pi, offers Raspbian as their official operating system. Raspbian is based on Debian, which in turn is a so-called Linux distribution [Sch14]. Linux is a project started by Linus Torvalds in 1991 who at the time was a student at the University of Helsinki. He wanted a free Unix system, but was not able to find one. So he started to work on his own implementation of a POSIX compatible Unix operating system [Lov10]. His work soon attracted like-minded developers and due to its licensing everybody was and still is free to contribute to the

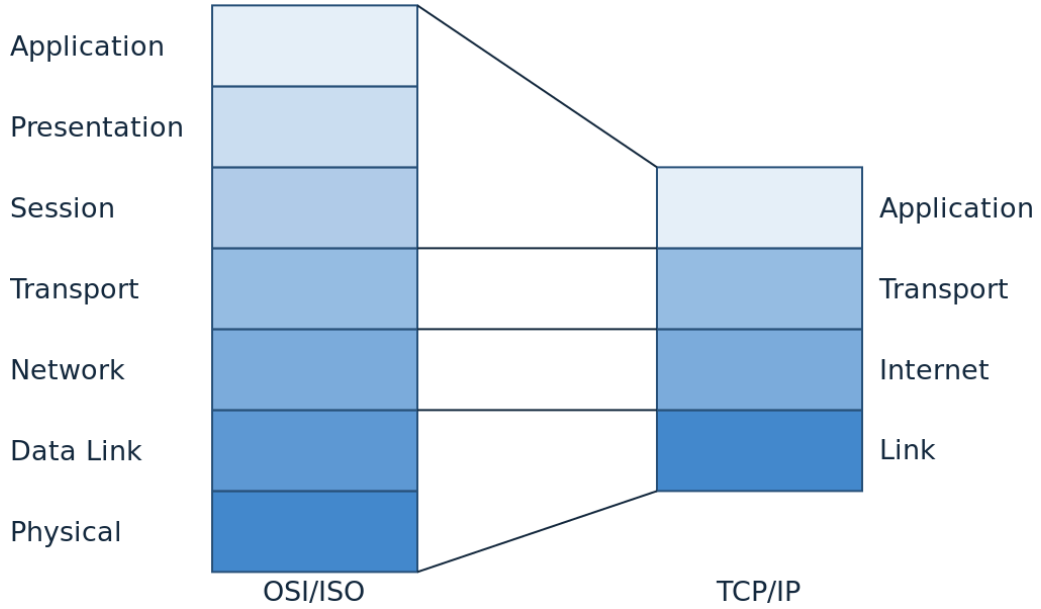


Figure 2.12: The TCP/IP model combines the functionality of the three topmost ISO/OSI layers into one single layer. The same applies to the lowermost two layers.

project. Today Linux is a complete operating system supporting many architectures [Lov10] and as mentioned before is the very core of the Raspberry Pi's official operating system [Har15].

2.4.2 Cross platform communication protocols

We shall now direct our focus towards the protocols needed to allow us to communicate between the game engine and the sensors and actors of the Jumpcube project. One prominent protocol suite is the TCP/IP stack, sometimes also referred to as the Internet protocols [DS11]. Differently from the ISO/OSI model, the TCP/IP stack only uses a four layer model instead of a seven layer one by collapsing the top three layer into a single Application layer and the bottom two into a single Link layer [IET89]. The Network layer from the ISO/OSI model is called Internet layer in the TCP/IP model [DS11]. This is depicted in Figure 2.12. Within this chapter the names of the layers shall be read in the context of the TCP/IP model.

Link layer

Starting at the bottom, the TCP/IP stack does not specify a certain protocol for its lowermost Link layer, but it does require from it that IP addresses are understood and translated from/to whatever addressing is used by the Link layer [Hun02]. The protocol employed in IPv4 is the Address Resolution Protocol (ARP), which allows to map IPv4 addresses transparently to the corresponding hardware addresses used for data

transmission [FS11]. It shall be noted, that the more recent IPv6 does not use ARP, but the Neighbor Discovery Protocol, a part of ICMPv6 which we shall discuss later in this section [FS11].

Internet layer

One layer above we find the Internet layer. Following RFC 1122 [IET89] the minimum required implemented protocols for the Internet layer and their respective functionalities are [FS11] [Hun02]:

- *Internet protocol (IP)* is the protocol used to transport data. IP is unreliable, which means packets are delivered in a best-effort manner and no guarantees are made that a packet is successfully delivered. In addition IP is connectionless, which means that IP does not maintain any state information about the packets or their ordering.
- *Internet Control Message Protocol (ICMP)* is used to transmit errors or other conditions. It employs IP as a transport protocol. ICMP is usually not used by programs directly, except for diagnosis tools like *ping* or *traceroute*. Yet, an error transmitted over ICMP may trigger an error handling routine in a program. As mentioned the IPv6 version of ICMP known as ICMPv6 also replaces ARP.
- *Internet Group Management Protocol (IGMP)* is employed to let other parties on the network know which hosts currently belong to which multicast group.

Transport layer

In the Transport layer we find the *User Datagram Protocol (UDP)* and the *Transmission Control Protocol (TCP)*. While both are concerned to provide data transport to the top layer, the kind of transport provided by them differs vastly. UDP provides no reliability, which means it does not take any additional measures to assure a packet reaches its destination. It only adds a header containing both source and destination port, the size of the so called datagram and a checksum [Hun02]. TCP on the other hand is a connection-oriented and reliable protocol. *Connection-oriented* means, that two parties on the network (usually referred to as server and client) need to establish a connection before they can start with the exchange of data [Ala14]. *Reliable* means that TCP takes additional measures to assure each packet given to it is correctly received by the other party. This is achieved by the following additional steps [FS11]:

- Data is broken up into segments. TCP decides how big a segment shall be.
- When TCP receives data from the other party it sends an acknowledgement.
- If a segment is not acknowledged in time the segment is resent.
- Each segment contains a checksum of its header and data. If a received segment contains an invalid checksum it is silently discarded. Because of the above rules it will eventually be resent.
- The ordering of the segments is preserved.
- Additional flow control is implemented to avoid buffer overflows.

UDP has less overhead then TCP as it only adds four header fields to the the data presented to it before handing everything over to IP when sending. When receiving it strips the four headers and passes the data on to the Application layer [Hun02]. In addition, being connectionless, data can be immediately sent without the need for explicitly establishing a connection [FS11]. However neither the server nor the client has any possibility to notice whether a packet is received by the other party or if it was lost without any additional measures. TCP on the other hand offers a more reliable communication channel than UDP by implementing additional measures to assure packets are correctly delivered. Yet, this adds overhead to both the amount of gross data transmitted [Ala14] and the latency, as a connection needs to be established before data can be exchanged. So in conclusion we may say that if reliability is required and of importance by the application it is better to choose TCP. If the application is not depending on every packet reaching its destination and it can handle a reasonable package loss then UDP is the better choice. Though implementing reliability in the Application layer on top of UPD is possible [FS11].

Application layer

RFC 1122 from 1989 [IET89] lists Telnet (remote login), FTP (file transfer) and SMTP (electronic mail delivery) as the most common Internet protocols. A very important protocol in the TCP/IP Application layer is the *Hypertext Transfer Protocol* (HTTP), which has become the primary protocol of the web [FS11]. Because of this fact HTTP is not only supported by Internet browsers, but also by many other applications including the game engines Unity [Unic] and Unreal Engine 4 [Gama], as well as the Arduino [Ard]. HTTP was designed as a stateless protocol and was developed for distributed, collaborative, hypertext information systems [IET14a]. HTTP is a stateless request/response protocol. The base information entity in HTTP is an Octet, which is any combination of eight bits. From these Octets messages are formed, which are exchanged by the two parties. Such a message may either be a request or a response [IET14a]. HTTP also requires a reliable transport protocol [IET14a] such as TCP. For identifying resources on a host HTTP in is current version relies on Uniform Resource Identifier (URI) as defined in RFC3986 with the following syntax [IET14a] [IET05]:

```
http-URI = "http:" "//" authority *( "/" segment ) [ "?" query ][ "#" fragment ]
authority = [ userinfo "@" ] host [ ":" port ]
```

Above, a segment is a sequence of allowed characters and character encodings as defined in RFC3986 [IET05]. HTTP uses so-called methods to indicate the purpose of a request. In total eight methods are defined in RFC7231, but only HEAD and GET are required by the specification [IET14b]. A complete list of the methods and their purpose can be found in Figure 2.13.

In 2011 the Internet Engineering Task Force (IETF) standardized the *WebSocket* protocol in RFC 6455 [IET11]. The aim was to provide a bidirectional communication channel between a server and a client, which is fully compatible with HTTP and its infrastructure.

Method	Description	Sec.
GET	Transfer a current representation of the target resource.	4.3.1
HEAD	Same as GET, but only transfer the status line and header section.	4.3.2
POST	Perform resource-specific processing on the request payload.	4.3.3
PUT	Replace all current representations of the target resource with the request payload.	4.3.4
DELETE	Remove all current representations of the target resource.	4.3.5
CONNECT	Establish a tunnel to the server identified by the target resource.	4.3.6
OPTIONS	Describe the communication options for the target resource.	4.3.7
TRACE	Perform a message loop-back test along the path to the target resource.	4.3.8

Figure 2.13: In total eight HTTP methods are specified, which allow to implement full CRUD (create, read, update, delete) access to a resource.
[IET14b]

Being part of the HTML5 standard WebSockets work in most modern browsers [Lub11] and thus are supported by a wide range of devices. Lubbers found, that HTML5 Web Sockets can reduce unnecessary HTTP header traffic by 500:1 and reduce latency by 3:1 compared to HTTP [Lub11].

However, being a new standard WebSocket bares the problem of backward compatibility with existing software [CHHR17]. For example, Internet Explorer up to including version 10 does not provide support for WebSockets [Rai13]. This problem is addressed by the *socket.io* project. It provides a full-duplex communication channel between a server and a client. WebSockets are used where available, but *socket.io* includes a variety of fallbacks for scenarios where WebSockets are not available. These fallbacks include for example Flash, XHR long polling, or JSONP polling [Rai13]. *socket.io* provides a more feature-rich and event-based API than WebSockets with support for [Rai13]:

- namespacing of messages
- multiplexing of connections
- disconnection detection
- reconnection
- broadcast messages

Similar to HTTP, *socket.io* is widely supported and implementations exist for Unity [Pan], Unreal Engine 4 [Kan], and Arduino [Roy].

In this chapter we have seen what hardware is available for building custom devices for the Jumpcube. We have also analyzed both the hardware and the software which allows

us to let said devices communicate with the game engine. This should allow us to provide an interactive multi-sensory virtual reality experience. In the next chapter we will first try to describe the challenges we are facing and then analyze what concrete frameworks and hardware components are at our disposal for solving these challenges.

CHAPTER 3

Methodology and design

In this chapter we will discuss the steps taken to build a multi-sensory system for use in Virtual Reality and other computer graphics applications. We will start by analyzing the problem. Next, we will define both functional and non-functional requirements for the system, which shall make sure it works as intended in a real time computer graphics application. Finally, we will introduce the components used during the development, first from the software perspective and then from the hardware perspective.

3.1 Problem description

The Jumpcube aims to bring a multi-sensory Virtual Reality experience to the user. As we have seen before such systems have already been employed in the realm of psychotherapy. With the development of the current generation of Virtual Reality headsets the visual sense can be seen as successfully addressed, as the companies currently predominantly seem to focus on further increasing the resolution of the displays. This in turn puts a higher demand on the computing power, which will need to be satisfied by the chip manufacturers. The auditive sense is also reasonably taken care of by game engines and their capability to produce a 3D audio experience on stereo headphones.

But commercially available systems like the Oculus Rift, the Playstation VR and the HTC Vive focus on visual and auditive sense only. Addressing senses beyond visual and auditive might bring some benefits though. As we have seen in Section 2.1.5 the widely accepted hypothesis attributes the so-called simulator sickness to contradicting sensory information. If we want to alleviate the effects of simulator sickness on users of the Jumpcube, we need to create devices which are able to generate stimuli beyond the visual and auditive. Since the Jumpcube is designed to be a interactive Virtual Reality system these devices cannot operate following a predefined schedule of events, but have to be controllable in real time by the simulation. Apart from impacting simulator sickness, we also expect the system to positively influence presence perceived by the user.

While looking for components allowing us to create such a multi-sensory system for computer graphics applications, we found that such systems have been developed in other projects. However, most of them fit a very narrow range of usecases and target enthusiasts. One example is the N1 M4A series from RSEAT Sim Racing Equipment, which aim to bring up to +/- 1g of g-force to the user, yet mentioned systems start at 16,999 $-US, compared to 1,099 $-US for the same product without the g-force simulating capabilities. The author was able to try a similar seat in conjunction with the *Assetto Corsa* racing simulator and the Oculus Rift DK2 and was truly impressed by the experience. Yet, being a seat, this product is not suited for building different experiences like e.g. a parachute jump simulation.

Worth mentioning is also Ubisoft's Nosulus Rift, a device which lets the user smell the virtual environment he/she is in. Yet due to its size and form factor the Nosulus Rift can hardly be used as intended with current generation Virtual Reality headsets. Furthermore, it never became a consumer product, hence cost remains purely speculative. In addition, the Nosulus Rift seems to have been developed for the sole purpose of marketing Ubisoft's *South Park: The Fractured But Whole* video game as it is up to date the only program known to support the device.

Based on these findings the author concluded, that it was inevitable to design and build multi-sensory devices from scratch to meet the requirements of the Jumpcube. The reason lies in existing devices being either not available on the market - including those which seemed promising when presented to the public - or did not provide the features required by the Jumpcube and - even if they would - showed to be too expensive.

Switching the focus towards how to integrate the devices with computer graphics applications, the author was able to determine standards for sensors and actors in conjunction with game engines, which are widely adopted by the industry. As an example the USB human interface device class shall be named, which aims to provide standardized support for keyboards, mice, and game controllers on the USB interface. DirectInput and XInput, both subsets of the DirectX API by Microsoft, do additionally provide support for haptic feedback. Yet it remained unclear to the author if and how these could be expanded or exploited to support feedback other than haptic as well. They also seem to be restricted to be used via USB, at least it remained unclear to the author if other physical connections could be used.

Another class of devices which looked promising for integrating multi-sensory deices into a computer graphics environment was home automation equipment, or *smart home* devices as the manufacturers like to advertise them. Nowadays, these come in a wide variety of form and function from simple switches and light bulbs up to garage doors and window blinds. But the devices based on the wired EIB bus tend to be relatively costly compared to the provided functionality, while cheaper kits rely purely on wireless communication protocols. Not only would this influence the resilience against environmental conditions like a polluted electromagnetic spectrum, but additionally a wide variety of standards

exist, like Z-Wave [1], Zigbee [2], Bluetooth low energy [3], Thread [4], and Wi-Fi [5], just to mention what seem to be the most common ones. The lack of an industry-wide standard inevitably either increases the effort necessary to implement the system as multiple standards have to be implemented, or bears the risk of a vendor lock-in, which in turn may affect future extensibility of the system as a whole. Finally, all of these devices are intended for home automation. This does bring the need for reliability, but not necessarily for the levels of latency required by a real-time computer graphics application.

Overall, the systems seem to have at least some restrictions. The lack of a standard for integrating especially actor devices into a computer graphics environment pushed the decision to try and implement one, which in the end should be able to accommodate all needs of the Jumpcube, while still providing extensibility for other kind of actors employed in projects similar to the Jumpcube. In the next section we will discuss what requirements need to be fulfilled by such a system. We will look at the features required from the hardware, and we will define the specifications to which the software shall adhere to.

3.2 System requirements

As we have seen, bringing a multi-sensory experience to a Virtual Reality environment is not a trivial task. Even though the system presented in the scope of this thesis mainly focuses on the needs of the Jumpcube, it shall still be flexible and standardized to a certain degree, so that it can potentially be used or extended for similar projects too. This flexibility adds some non-functional requirements to the already present functional requirements. We will first look at the latter, followed by the former.

3.2.1 Functional requirements

Since multiple senses need to be addressed, and the setup continues to evolve in the course of the project, the system is required to support multiple actors. At project start the first requirement was to add wind to the first of the Jumpcube's simulations, i.e. a parachute jump over the city of Vienna. The wind effect was chosen, as it was recognized by the team to be the most obvious haptic stimulus during a parachute jump. To enhance the feeling of acceleration during the jump, the wind should also adapt to the speed the user moves through the virtual environment, so a simple on-off switch was not feasible. Over the time more stimuli were added, including smell, heat, moisture, and g-forces. Hence, the system does not only need to support multiple devices, but also multiple types of devices. The latter bears the need for a flexible design, since some of these devices have to communicate with or control existing hardware.

[1] `http://www.z-wave.com/`, last visited 16th August 2018, 15:45 CEST
[2] `https://www.zigbee.org/`, last visited 16th August 2018, 15:45 CEST
[3] `https://www.bluetooth.com/`, last visited 16th August 2018, 15:48 CEST
[4] `https://www.threadgroup.org/`, last visited 16th August 2018, 15:50 CEST
[5] `https://www.wi-fi.org/`, last visited 16th August 2018, 15:51 CEST

From a software perspective the system has to be able to communicate with different game engines. The minimum requirement is support for Unity and Unreal Engine 4, as they come with a free-to-use licensing model for non-commercial projects. Unity was the engine of choice in the beginning of the project, as some members of the initial team already had experience in working with it. Unreal Engine 4 on the other hand is currently one of the best engines on the market and widely used in the game industry too. Even though it took some time until the first simulation based on Unreal Engine 4 came to the Jumpcube, the support for it was a requirement from the beginning.

Being part of an interactive simulation brings additional requirements to the system. The most obvious one comes from the interactive nature of the Jumpcube. This means by definition, that event timings are not known a priori. In other words: decisions by the user may affect when - if at all - a user triggers an event, which in turn requires an actor to react. Moreover, the latency between an event being triggered in the simulation and a reaction from the corresponding actor has to be within a reasonable margin. Current Virtual Reality headsets like the HTC Vive operate at a refresh rate of 90 Hz. This means, that in a best-case scenario 90 frames can be presented to the user within one second. This also means each frame spans over 11.i̇ milliseconds. Since we can neglect a delay of one frame, we require from the system to react within 10 milliseconds. Further hard real-time guarantees are not explicitly required from the system.

Additionally, the communication must not have any noticeable impact on the frames-per-second delivered by the simulation. Since the logic needs to know the internal state of the simulation (like e.g. the users velocity for the adaptive wind intensity mentioned above) the system has to be part of the rendering process at least to some degree. Because of the way how the scripting API in Unity works, long lasting computations directly affect the frame rate. To avoid any impact, the API used in the game engine shall be asynchronous and event based to interfere with the rendering procedure as little as possible.

Another desired functionality arises from the Jumpcube's mobile nature. The Jumpcube is intended as an attraction at public or semi-public events, like conferences, trade shows, or science fairs. Hence, it is not a static structure to be built and never touched again; on the contrary, it can be dismantled, moved, and reassembled if needed. This also affects the multi-sensory feedback system, as it needs to move together with the bearing structure and it too needs to be dismantled and reassembled. For this reason any needed wiring shall be as uniform as possible. In addition, standardized cables shall be used wherever possible. In the long term this should reduce the risk of a complete outage of the system or parts thereof because of a defective cable, which cannot be replaced due to the absence of matching replacement parts. Finally, the cables shall support easy plugging and unplugging while still providing reasonable protection against accidental or stress-based unplugging. Since some of them might need to be wired through tiny gaps or narrow bends, they shall be flexible and durable.

To further prevent long lasting outages because of failed parts, the system shall be engineered in a way which allows to easily troubleshoot any connected hardware and any cabling. This means that if for example a cable breaks - a condition, which cannot

be avoided - the overall status of the system needs to be analyzable without the need to dismantle it. In such a scenario it shall be possible to locate the failed cable with the help of the gathered overall system status and replace it within a reasonable short maintenance interval. In addition it is desirable that the failure of a single hardware component does not cause a complete system outage. A single point of failure cannot be excluded in the Jumpcube because the PC running the simulation will always be one. But the multi-sensory feedback system shall add as few additional single points of failure and shall provide the capability to continue working in case of a reasonable amount of hardware outages, possibly in a degraded mode. It shall also resume full functionality once the broken parts have been replaced with little to no additional user interaction.

Finally, the system shall provide an API to devices not being part of the core system. This may be a laptop, a tablet, or even a mobile phone. Via this API the operator shall be able to examine the current state and to control all features of the system. It is desirable to have a graphical user interface, which in addition to showing all available features at a glance also enables the operator to trigger certain events or to change the parameters of the system with as littles effort as possible. This user interface shall be optional for the operation and further must not affect the availability of any given part of the system in an undesired way under any circumstances. An exception may be emergency shutdowns, which in this scope are regarded as desired.

3.2.2 Non-functional requirements

Apart from the functional requirements we identified a series of non-functional ones, which are crucial either for the project to succeed in the first place, or for its later development and potential commercialization. Even though the latter is not a primary goal it shall be taken into consideration. The first of these requirements are the costs. The system shall be built and maintained with reasonable spendings. To achieve this it is desirable to use off-the-shelf hardware components wherever possible and to avoid custom hardware. Said measures also affect the second non-functional requirement, namely the possibility to reproduce the system either at a later time, or within a different project. Though, additionally to off-the-shelf hardware we need a second factor for reproducibility, which is long-term support of both the used software and hardware. For that, the system shall either use software and hardware with guaranteed long-term support or alternatives, which have existed for a long enough time to reasonably assume they will be available in the near and mid-term future. It shall also be noted, that the restriction to a certain software platform or programming language shall be avoided in order not to tie the possibility to reproduce the system to the availability of said platform or language.

To further avoid any risk depending on the availability of a certain product - be it software or hardware - the system shall be kept as modular as possible. By avoiding a so-called vendor lock-in, in case of the unforeseen end-of-life of a component, the system should be able to be adapted to a similar replacement of said product by only applying some local changes, while keeping the overall system mostly untouched. Finally, by the use of open standards for communication and open source components for the hardware

and software this risk can further be diminished, as the probability of an open product suddenly disappearing without any replacement seems smaller, than the same scenario with a proprietary and possibly not publicly documented standard. Furthermore, if it does reach its end-of-life, open source components and open standards can potentially still be maintained or manufactured by any individual - a possibility which in general is not available with proprietary components.

The focus on open standards and open source components not only helps with the long-term reproducibility as discussed above, but may also positively affect a potential launch to market. Open source components may bear some restrictions of use in their licenses. As an example: the General Public License, one of the most common open source licenses, demands that any change made to the subject of the license has to be made accessible by the user thereof. But the components covered by open source licenses are free to use and even though any changes made to them may have to made public, they offer the possibility to change and adapt them to ones needs. These possibilities too are not available with proprietary components in general. Further on, open source components mostly come without any patent fees or royalties attached. This could become a significant advantage during a potential launch to market.

3.3 Available frameworks

In addition to the features mentioned above, relying on open standards and open source software also brings the advantage of a broad availability of existing frameworks for both software and hardware. In this section we shall analyze the available software components to ease the development and allow for rapid prototyping of our multi-sensory devices.

The decision for any given component of the software stack was made based on the following requirements:

- *Sustainability:* It can be safely assumed the component will be available in the near to midterm future, because it is being actively developed, an active community has formed around it, and/or it is widely supported and/or used in the industry.
- *Portability:* Established programming languages and standards are used to reduce the effort necessary to port the system onto a new platform. This should also reduce the training needed for a new member to start working on the project.
- *Efficiency:* The component is well-documented and allows for rapid prototyping to minimize the risks caused by the tight schedule and the hard deadline of the project.

In addition, the decisions for the chosen technologies influenced each other, which renders it difficult to rank them. Hence, the following order is essentially arbitrary.

Listing 3.1 SocketIO communication front end using Flask

```
1  from flask import Flask
2  from flask_socketio import SocketIO

4  app = Flask(__name__)
5  socketio = SocketIO(app)

7  @socketio.on('some_event')
8  def do_something(message):
9      pass # Business logic may be called here

11 socketio.run(app, host='0.0.0.0')
```

3.3.1 Web application frameworks

Since time was a scarce resource at project begin and the deadline, being the 200 years festivities of Technische Universität Wien, was sharp, the need for rapid prototyping arose. This meant for the author, that the best possible implementation would be the one which both works reliably and can be implemented in time. Hence, the decision was made to rely on the Web technologies, since support for them exists within the game engines as we have seen before and the author already had some experience in working with them. This also meant the idea of basing the communication backbone on fieldbus technology - even though it may be a more adequate solution - was dropped in favor of a TCP/IP based one for lower development costs and time. For the same reasons the *C/C++* programming language, which - as far as the author is concerned - would have been the best choice was dropped in favor of *Python*.

We chose the *Flask* [6] framework as the base for our implementation on the devices, as it is lightweight, open source, can be based on top of the high performance *gevent* event loop, and hence leverages the performance of the C language from within Python, and it facilitates the separation of the communication front end from the business logic. Flask also offers a library, namely *Flask-SocketIO*, which allows us to integrate it with SocketIO. A minimal implementation of a Flask based SocketIO front end can be found in Listing 3.1. We can further exploit the fact that Flask allows to implement a SocketIO server so that each device represents its own server. This does not limit the connectivity, as each device can still communicate to each other by connecting to it via a SocketIO client. But it does improve the system reliability, since the outage of a SocketIO server only affects that very device, while all others remain operational.

[6] http://flask.pocoo.org/, last visited 16th August 2018, 15:39 CEST

Listing 3.2 SocketIO client using reference implementation

```
1    var socket = io.connect('http://' + document.domain + ':' + location.port + '/');
2    socket.emit('some_event', 'Hello World');
```

3.3.2 User interface

Basing the communication backbone on TCP/IP also allows us to use an HTML page as a graphical user interface for the system. Since SocketIO is a technology designed for the Web, adding support for it on an HTML page can be accomplished by the reference JavaScript implementation. A minimal client for the front end defined in Listing 3.1 can be found in Listing 3.2. This approach enables us to access the graphical user interface from any device able to run a fairly modern web browser. Since web browsers have different base style sheets, relying on plain HTML for the graphical user interface may lead to an inconsistent look and feel for different browsers. To avoid this we used the *Materialize* [7] CSS framework. Apart from delivering a consistent look and feel, Materialize also aids in the development of layouts, which are able to adapt to different screen sizes. Finally, Materialize also features widgets like switches and sliders optimized to be comfortably usable both with mouse and keyboard, as well as via a touchscreen.

3.3.3 Arduino

For time critical tasks we employ microcontroller based Arduinos. For programming them we shall use the *Arduino IDE* provided by the Arduino project team. This is not the only way available for programming an Arduino. For example Makefiles exist, e.g. the *Arduino-Makefile* available on GitHub, which implement the full workflow from compiling the sources to flashing the created binaries onto the Arduino board using the popular *Make* build automation tool. Yet the Arduino IDE, apart from supporting a wide range of boards, features a graphical IDE and some built-in debugging aids like a serial console. For communicating over SocketIO using an Arduino with Ethernet shield the *socket.io-arduino-client* available on GitHub is at our disposal. Though, we fear an Arduino does not have the computing power to act as webserver for delivering the graphical user interface to connected clients and mentioned library only implements a SocketIO client. However, the system as a whole is more resilient if each device is a SocketIO server. Therefore, communicating to the Arduino via SocketIO is sub-optimal and may not be adequate for productive use. As a consequence we will use a Raspberry Pi as SocketIO server, which also delivers the graphical user interface, and communicates with the Arduino over a USB connection. Experiments with an Arduino, an Ethernet shield and said library were conducted as proof of concept for such an implementation.

[7] https://materializecss.com/, last visited 16th August 2018, 15:43 CEST

3.3.4 Yocto Project

As a first topic we shall look at the *Yocto Project* [8], or Yocto for short. Yocto is at its core a fully featured build environment for a Linux distribution. Yocto allows to create a minimal, custom tailored Linux distribution for a device. It achieves this by not only providing the source files, but also a complete compiler suite and toolchain, allowing the user to build the distribution for a variety of devices, even if they are based on different CPU architectures. In addition Yocto allows to compile for a hardware architecture different then the one the compiling machine is based on, also known as cross compilation, out of the box. The most bottom entity in a Yocto project is a so-called recipe. A recipe is an abstraction of a build script written for the BitBake build system. In other words, a recipe defines where the source code resides, how to fetch it, which patches shall be applied to it, and what to do during the various steps from download to installation. A recipe also allows to define dependencies to other software packages for both build-time and runtime. However, the user does not need to implement the full recipe, as Yocto already provides a variety of templates for the most common build systems and software distribution system. As an example Listing 3.3 shows a recipe for a Python package available at the Python Package Index (PyPi).

Multiple recipes can be bundled together in a so-called *layer*. A layer may have different responsibilities, such as adding support for a specific hardware platform, provide build instructions for specific software packages, provide templates for recipes, or any combination thereof. Hence a Yocto based build environment mostly will contain multiple layers. In this project additionally to the base layers providing the toolchain and the compiler we will use the *meta-raspberrypi* layer to add support for the Raspberry Pi single board computer, and the *meta-openembedded* layer which provides recipes for software packages such as Python and OpenSSH. Using Yocto allows us to build deployable images for a variety of hardware platforms without the need to change the build instructions in case a new hardware platform needs to be supported. In addition, since the used layers and recipes are distributed via the Git version control system, the builds are fully reproducible on every machine running a fairly recent version of a Linux distribution, even across different devices with just the knowledge of the revisions of the individual layers. This lowers the risk of loosing important data because of a broken storage device, as a build can easily be reproduced. Finally, using Yocto we have a very fine grained control over which software will run on the device, which ultimately lowers the risk of unwanted side effect caused by an unneeded piece of software running on the device. BitBake also offers the ability to extend or alter any aspect of an existing recipes via so-called `bbappend` files.

3.4 Available components

Having seen what software components are available to us we shall now have a look at the hardware components at our disposal. Each of the following sections describes one type

[8] `https://www.yoctoproject.org/`, last visited 16th August 2018, 15:42 CEST

Listing 3.3 File **python-flask-socketio_2.9.2.bb**

```
1   SUMMARY = "Socket.IO integration for Flask applications"
2   LICENSE = "MIT"

4   LIC_FILES_CHKSUM = "file://LICENSE;md5=38cc21254909604298ce763a6e4440a0"
5   RDEPENDS_${PN} = "python-flask python-socketio python-gevent-websocket"

7   PYPI_PACKAGE = "Flask-SocketIO"
8   SRC_URI[md5sum] = "1f8521101d2c9b4155cf521fbce3740c"
9   SRC_URI[sha256sum] = "0fb686f9d85f4f34dc6609f62fa96fe15176a6ea7e6179149d319fabc54c543b"

11  PR = "r10"

13  inherit pypi setuptools
```

The various parts explained:

- the file's name sets the default values for the package name `PN` and the package version `PV` in that it consists of `PN_PV.bb`
- `SUMMARY` and `LICENSE` are mostly informational
- `LIC_FILES_CHKSUM` contains the file describing the license and its MD5 hash. This way, shall the license ever change, the build will fail and the user can revisit the new licensing conditions to assure they are still compatible with the project.
- `RDEPENDS_${PN}` defines the runtime dependencies. Build-time dependencies would be given using `DEPENDS_${PN}`.
- `SRC_URI[md5sum]` and `SRC_URI[sha256sum]` contain the MD5 and the SHA256 hash respectively of the downloaded package. This way download errors and manipulations on the package can be detected.
- `inherit pypi setuptools` instructs BitBake to use the templates for PyPi packages to be installed via setuptools. This combination is a very common way of installing packages on the Python platform.
- `PYPI_PACKAGE` is specific to the `pypi` template and identifies the package to be fetched. It defaults to `${PN}` without the `python-` prefix. Since it is convention in BitBake to have all lowercase package names we need to override it here.
- `PR` is the package revision. Since Yocto heavily caches build artifacts, increasing this number is an easy way to force a rebuild in e.g. the case the recipe changes.

of sensory feedback. Overall, they are ordered chronologically by their first appearance within the project.

3.4.1 Communication backbone

Having chosen the TCP/IP stack and the respective protocols as our means of communication, the choices for the communication backbone are fairly restricted. As a main communication hub we chose an off-the-shelf 10/100 MBit/s Ethernet switch. A switch makes sense in our scenario, as it allows full-duplex and hence collision free communication. All of the PCs and the Raspberry Pis used in the setup come with a built-in Ethernet adapter. We used random UTP Ethernet cables, since all we tested seemed to work with 10/100 MBit/s just fine.

If the system shows any unexpected behavior which may be caused by an overloaded network, we still have the option to move to a 1 GBit/s switch. The PC used for the project is capable of running 1 GBit/s Ethernet, and since most of the communication is expected to happen between PC and the switch, we might see an immediate increase in performance. Yet, not even the most up to date Raspberry Pi 3+ is able to exploit the full potential of a Gigabit Ethernet link, as its Gigabit network adapter is connected to the SoC using a USB 2.0 connection, which allows a maximum of 480 MBit/s gross data throughput. It shall be noted, that other single board computers exist, which are able to exploit the full potential of a Gigabit Ethernet link, but these devices tend to be higher priced then the Raspberry Pi. Because we use Yocto and some devices with Gigabit connectivity are supported by it, we should be able to switch to one of them with ease should it turn out the Raspberry Pi is not powerful enough. In such a scenario we also need to check all cables to see if they support Gigabit Ethernet connections and introduce CAT5e UTP or higher rated cables for those which are not.

3.4.2 Wind

Since the first application developed on the Jumpcube was a parachute jump towards Vienna, it was the obvious choice to implement wind simulation at first. As already mentioned, it was required that the wind speed shall be bound to the velocity of the user's avatar within the virtual world. This meant we needed to find some fan configuration which is controllable via software. We identified two ways of achieving this:

- Direct current (DC) fan with pulse width modulation (PWM) control
- Alternating current (AC) with upstream AC dimmer

We chose the latter, because of the following reasons:

- availability and cost of fairly powerful 230 V AC fans
- no need for additional AC/DC converters
- reusability of the 230 V AC dimmer on later features

It shall be noted though, that the use of a high-power DC fan with PWM control input constitutes a viable approach and offers other advantages. But since flexibility, reusability,

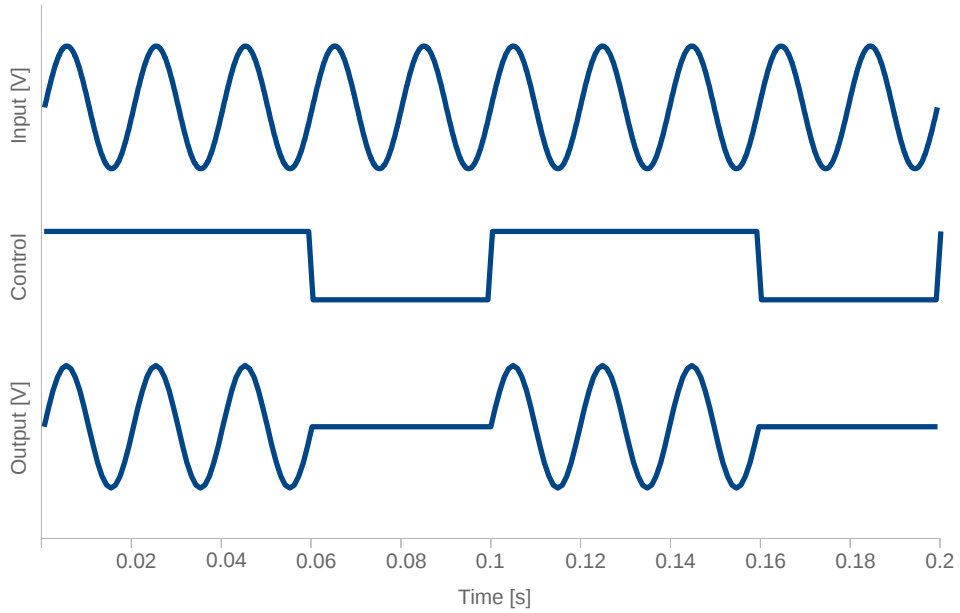


Figure 3.1: Zero crossing based PWM control of an AC motor. AC voltage from power plug (top), control signal (center), and output AC voltage (bottom) of the fan controller running at level 6.

and simplicity is of high importance in the case of the Jumpcube we opted against it. Finally, the system as a whole could be easily adapted to support PWM control for a fan or other periphery, but because of the lack of such peripherals it has not been implemented up to this date.

Power-controlling a 230 V AC fan turned out to be more of a challenge then expected. Voltage controlling by a electronically controllable transformer turned out to be too expensive and even after extensive research on the topic it remained unclear, if the chosen fans would respond to it as expected. During said research the author found numerous sources ruling out trailing-edge phase-fired control, since AC motors are inductive and hence the power cut-off on the trailing edge may cause a voltage spike due to the instant collapse of the magnetic field in the motor's coils. These voltage spikes may cause serious damage to the electronics and may be dangerous to humans as well. Circuits can be prepared to handle such scenarios gracefully as we will see in Section 3.4.4, but due to the black-box character of off-the-shelf 230 V AC fans we ruled out this approach as too risky. During the research we found no mention of leading-edge phase-fired control in combination with 230 V AC fans. When trying it though, the fan only produced a crackling noise and its blades did not turn.

As final solution we chose power switching on zero crossing. To understand how this works, one needs to know how current from a standard 230 V AC European power socket works. The voltage oscillates in a sine wave at 50 Hz. Since a sine wave has two

zero crossings per period this leads to the voltage having 100 zero crossings per second. Switching the fan at exactly these zero crossings turned out to work fairly well. Yet, this approach has the drawback of only being able to switch the fan on or off at most 100 times a second, being once in 10 ms. At least for fan control though, this did not turn into a major disadvantage, as the inertia of the fan's blades was much more relevant overall than the switching rate. We decided to toggle the fan's power state twice in 100 ms to reduce the latency at the cost of having less power levels. The fan is turned on at the beginning of each 100 ms interval, which needs to be exactly at a zero crossing and should in this context be counted as the zeroth of its kind. The fan is again switched off at the nth zero crossing to drive the fan at level n (e.g. level 6 means 6 half waves on, 4 half waves off). Figure 3.1 depicts a diagram of the fan controller operating at level 6. This lead to 10 potential switching points and hence an 11 step power control for the fans (off, 9 intermediate steps, full on). Level 0 (off) and level 10 (full on) involve no intermediate switching.

To be able to switch power fast and with a frequency of up to 100 Hz we decided to use solid state relays. While being more expensive than mechanical relays, they offer higher switching frequencies, less response time, and the can be directly driven by a low voltage and low current output like in our case a digital output of an Arduino. For detecting the zero crossings on the AC input we used a zero crossing detection IC connected to the interrupt input of the Arduino. The first power controller circuit board was designed and built by a co-worker in the project (Gordan Savičić) during a different project. It offered 8 output channels and was controlled by an Arduino Mega board. It worked and performed well, yet due to its round shape it was difficult to integrate in our setup. Another co-worker (Florin Hillebrand) later built a smaller version designed to fit in a 1U rack case. It offers 4 output channels, as we deemed that to be sufficient, and is controlled by an Arduino Nano v3. All credit for both the circuit board and software running on the Arduino Nano v3 goes to Florin Hillebrand. The schematics, as well as the software are available on his GitHub profile [9]. The Arduino is connected to a Raspberry Pi 2 via USB, which integrates it with the rest of the system and also provides a graphical user interface. The AC dimmer was later used to incorporate moisture, and heat into the system as well.

3.4.3 Centrifugal force

Simulating forces is crucial to the experience in Virtual Reality, and contributes to avoid simulator sickness as we have seen. For that reason, the second addition to the Jumpcube's multi-sensory system was a controller allowing to simulate centrifugal forces. This was achieved by connecting a servo motor to the ropes connected to the user's hips. This way we are able to rotate the user around his/her roll axis. We chose a servo motor from Swiss manufacturer Maxon Motors due to its high popularity. Maxon Motors provides a controller for their servos, namely the EPOS series. We used the EPOS2 24/5 model. The EPOS2 24/5 can be controlled via USB using a C library provided by the

[9] `https://github.com/flozzone/ssr-dimmer-board`, last visited 8th August 2018, 17:31 CEST

manufacturer, which is available for Linux on the x86 and the ARMv7 platform. Due to Python's capabilities of loading and using C libraries via the `ctypes` module we were able to integrate it with our Raspberry Pi 2 based controller setup.

For security reasons we put a clutch between our servo motor and the mechanics connected to the ropes leading to the user's hips. Though, this caused the servo's integrated position control to be unusable, as with the clutch open the servo would turn without moving the user and hence an offset would be introduced. Therefore we control the position of the mechanics using a linear 50 kΩ potentiometer. The potentiometer is connected to the analog input of an Arduino, which in turn is connected to the Raspberry Pi 2 which controls the EPOS2 24/5 device via USB. The Arduino also allows us to open or close the clutch from software.

The overall setup is controlled by a bang–bang feedback control which lets the motor either turn clockwise or counterclockwise. The current status of the system is evaluated using the potentiometer, the desired state can be set using the SocketIO API, either from within the game engine, or from the graphical user interface. Even though bang-bang controllers are fairly limited compared to more complex feedback controllers, it worked well in our setup and we settled with it. All credit for the additional mechanics and electronics for this part of the system go to project members Bela Eckermann and Florin Hillebrand.

For being able to further adapt the g-force simulation we integrated an inertial measurement unit (IMU) with the application. We chose the Invensense MPU-6500 for its feature set and its software support, while still being reasonably priced. The IMU would be positioned on the user's chest and is able to measure the roll angle. Attached to it we have an Ardunio Nano v3, which queries the IMU over I2C and relays the data to the PC over USB. The Arduino Nano v3 was chosen because of its small size, while having approximately the same feature set as a much larger Arduino Uno. The input can be used both to control the virtual avatar, and to instruct the servo motor to either support or counteract the user's motion around its roll axis.

3.4.4 Odors

Since simulating odors is a fairly exotic topic, we reached out to Scent Communication, a company specialized in producing artificial smells. Apart from providing the different scents used in our setup, they were also able to contribute a prototype able to control up to six different smells. The ScentController 6, built by Dräger, is controlled by seven electronic valves, one per scent and one so-called flush valve, which bypasses the scent cartridges and allows to pump clean air through the system, i.e. flush it. Since these electronic valves need 12 volt to operate we built a simple transistor circuit board, which allows us to to control the valves via a lower voltage and low currents as provided by e.g. the general-purpose input/output of a Raspberry Pi. Since electronic valves, similar to 230 V AC motors, are inductive, we needed to make sure the voltage spikes originating from suddenly turning off the valve would not harm the electronics. For that, each of the

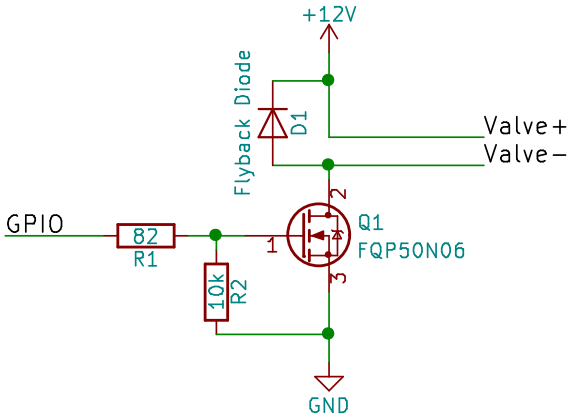


Figure 3.2: The control circuit for a single 12 V electronic valve. The control input (GPIO) is overcurrent protected by a resistor (R1). The FET transistor (Q1) allows to control a 12 V rail using a lower voltage and low current. The pull-down resistor (R2) is responsible for a defined potential on the transistor's gate in the off state. The flyback diode (D1) prevents damage from voltage spikes on switch off.

seven valve control circuits contains a flyback diode able to drain off voltage spikes. A schematic of the control circuit for a single valve can be found in Figure 3.2.

The seven valve control circuits are driven directly from the GPIO of a Raspberry Pi 2. Since we do not need strict realtime guarantees as e.g. when controlling the AC fans via the zero crossing mechanism, and the tube connecting the device's output to Virtual Reality headset causes a delay of around 0.5 seconds, we do not need an additional Arduino and any latency introduced by the Raspberry Pi's operating system is neglectable.

This concludes the chapter on methodology and design. The problem we are trying to solve lies in integrating multi-sensory devices with computer graphics applications suitable for use with Virtual Reality. We have analyzed the products currently available on the market and found none to be suitable for the usecases of the Jumpcube. We went on to define both the functional and non-functional requirements we impose onto such a system. Finally, we analyzed what software and hardware components are at our disposal for rapid prototyping of a system meeting those requirements from both a software and a hardware point of view. For the latter, we have listed and described the devices which will produce the multi-sensory feedback. For the sooner we concluded Web based technologies, i.e. SocketIO, HTML, JavaScript, CSS, and TCP/IP, in combination with Yocto and Python are to be preferred over a C/C++ and fieldbus based solution because of time and cost advantages. The drawback of the chosen technology stack lies in potentially inferior performance. We will now move on to show how the components were implemented. We will also discuss our benchmarks to see if the performance meets our specifications.

CHAPTER 4

Implementation

This chapter contains the technical details for both the communication backbone, as well as the individual controller devices. For the sooner we will also conduct some benchmarks to see if it is able to meet the requirements imposed upon it. For the latter we will document the provided APIs, their parameters and functions, and by which other parts of the system they shall be used.

4.1 Communication backbone

Our first task is to see, if the combination of TCP/IP, WebSockets, SocketIO and Python on the software side, and Ethernet, consumer grade networking equipment and Raspberry Pi 2 on the hardware side meets the functional requirements imposed upon it. Meeting these requirements is of utmost importance for the system, as a failure to do so will inevitably cause contradicting sensory information, and in turn may cause simulator sickness.

4.1.1 Testing methodology

The test setup involves two Raspberry Pi 2 Model B, referred to as *Alice* and *Bob*. Alice acts as a SocketIO client, while Bob constitutes the SocketIO server. Both are connected to each other via a TP-Link TL-WR841N, which features a 10/100 MBit Ethernet switch, using UTP cables. The latency is defined as the time between an action of Alice and the corresponding reaction of Bob. As actors the general-purpose input/output (GPIO) pins of the respective Raspberry Pi are used. The latency is measured using an Arduino, which measures the time between two interrupts, one being triggered by Alice, the other one by Bob. The measurements are then sent via USB Serial to a connected PC, which collects the data for further analysis. In addition, a two channel oscilloscope is connected to the GPIO pins of Alice and Bob to provide a sanity check for the data collected using the Arduino. The full hardware setup is depicted in Figure 4.1.

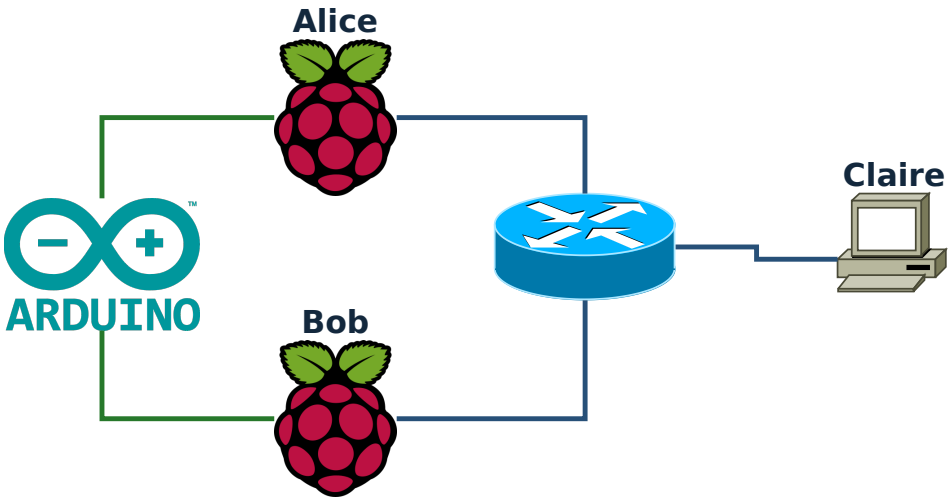


Figure 4.1: The benchmark setup. Blue lines represent Ethernet cables, green lines are electrical connections. Alice and Bob communicate via SocketIO. The Arduino measures the latency between the two using interrupts. Claire produces noise on the network to simulate network load.

The VR applications built for the Jumpcube feature a duration of approximately five minutes each. Afterwards the program is closed and restarted for the next user. This causes all communication channels between the simulation and the multi-sensory system to be closed and reestablished. A benchmark run should simulate one such iteration. We decided to add another two minutes, yielding a total benchmark length of seven minutes.

During a second benchmark run a loaded network is simulated by the addition of a PC referred to as *Claire* sending events to Bob in an endless loop. This is meant to simulate a worst case scenario, as during the operation of the system events will not occur more often then once per rendered frame. In addition, each of the devices constitutes a SocketIO server with the PC running the simulation being the only client. Half-way through the benchmark Claire shall be forcefully shut down to see if and how the system handles sudden connection aborts. Since for this benchmark we are not interested in long-term stability, but rather how the system behaves under synthetic load and sudden connection aborts, we decided to lower the benchmark runtime to four minutes.

Both benchmarks follow the same communication protocol. At the beginning of each benchmark the GPIO pins of both Bob and Alice are configured low. The communication adheres to the following protocol:

- Alice: set GPIO pin to high
- Alice: send event to Bob
- Bob: whenever an event is received:
  - toggle GPIO pin
  - wait for 10 milliseconds

  - send event to Alice
- Alice: whenever an event is received:
  - toggle GPIO pin
  - send event to Alice

It shall be noted that due to the event-driven nature of the described protocol every lost event would bring the benchmark to a sudden end.

4.1.2 Latencies

We found the latency to be 4341 µs on average with a standard deviation of 78 µs for the test on the otherwise idle network. The 0.99 percentile is 4564 µs. This is well below the targeted 10 ms. We also see that performance is stable over time, at least for time periods of approximately seven minutes. Figure 4.2 shows a plot of the collected values.

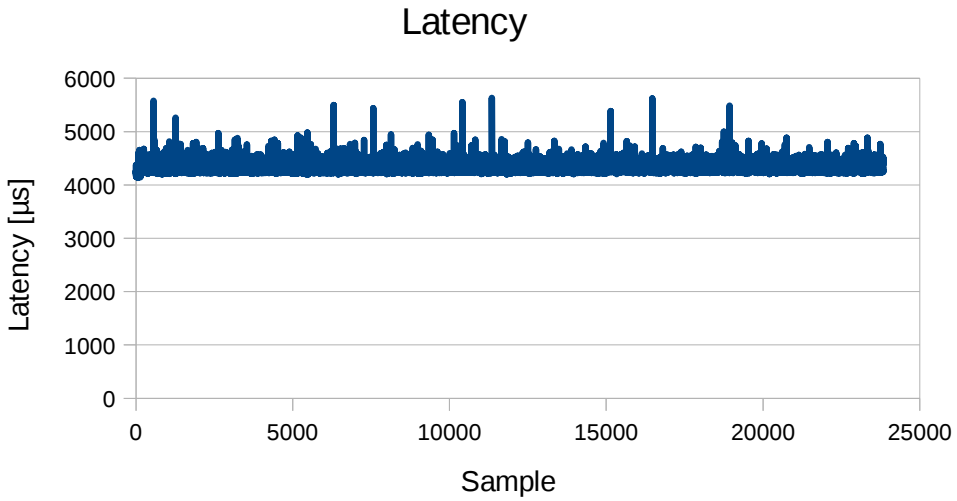


Figure 4.2: Latency over time on an otherwise idle network. The samples were taken over a period of seven minutes.

For the run on the loaded network we calculated the values for the whole run, as well as for both parts (i.e. with synthetic load and after Claire's shutdown) individually:

Period	**Average**	**Std. deviation**	**0.99 percentile**
Loaded	4788 µs	106 µs	5116 µs
Not loaded	4397 µs	120 µs	4735 µs
Overall	4589 µs	226 µs	5060 µs

Figure 4.3: Statistical evaluation of the measured latencies.

We see from the results that artificial load does have a negative impact on all metrics. However, even in the worst case scenario with synthetic load the values are still well below the required 10 ms. After shutting down the artificial load the average returns to be the

same by margin of error as for the benchmark on the otherwise idle network. However, the standard deviation is still higher than both during load and on the otherwise idle benchmark. We assume this phenomenon to be caused by the best-case latency being the same as on the otherwise idle network, though the worst case still being influenced by the previous load. These two factors could together cause the higher variance we see. Figure 4.4 shows a plot of the collected values. It shall also be noted, that the standard deviation in the *Overall* row, being the sum of the values for the two individual measurements, does not carry any relevant information, and is just listed for completeness.

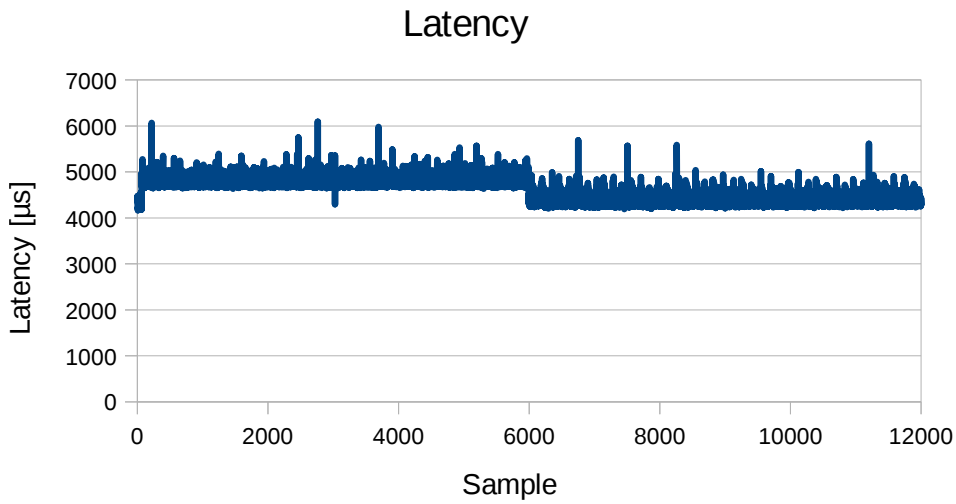


Figure 4.4: Latency with artificial load and after forceful shutdown thereof.

4.1.3 Conclusion

We have seen that the latency of the system is well below the required 10 ms, even in a scenario with synthetic load. We have also seen, that the system is reliable as no event was lost during neither of the benchmark runs. We have thus shown, that the system is adequate for computer graphics environments in the scope of Virtual Reality.

4.2 Environment controllers

In this section we will have a closer look at the individual controllers, their overall architecture, their APIs, and their respective functionality. In total, three different types of controllers were implemented to interact with three different kind of devices. At the current stage, due to the low number of actual controllers, we did not find the necessity to implement a discovery mechanism. Therefore, we rely on static IP addresses defined at build time via the respective BitBake [1] recipes. Nevertheless, the system could be extended to provide automatic discovery at runtime.

[1] `https://www.yoctoproject.org/software-item/bitbake/`, last visited 24th October 2018, 15:42 CEST

The types are (with their respective IP address in parenthesis):

- *Dimmer:* (192.168.1.50) Controls up to five 220V AC devices.
- *EPOS controller:* (192.168.1.51) Its main purpose is to interface with the Maxon Motor EPOS2 series servo motor controllers. This controller shows how to integrate with devices using a third-party library.
- *Vragrancer:* (192.168.1.52) Mainly designed to support odors, this controller is suitable for devices requiring a digital 3.3 Volt and low current control signal, or by extension a 12 Volt high current control signal. Moreover, other voltages and currents can be supported by the addition of different transistor circuits or solid-state-relays.

4.2.1 Common Properties

All controllers have some aspects in common. First and foremost is a HTTP webserver which runs on port 5000. This server is serving the WebUI, which can be accessed using any modern web-browser by navigating to `http://<IP>:5000`. Being a web technology, the SocketIO server responsible for the actual API can share the port with the HTTP server. It is recommended to connect to it using the WebSocket protocol via `ws://<IP>:5000/<namespace>`. All clients, including the WebUI, shall connect to the respective device using this method. The software itself is hardware agnostic. The only requirements are IP connectivity for the HTTP and SocketIO parts of the API, and the ability to run Python code. But, since SocketIO is an open and widely supported protocol, reimplementations or extensions of the system may use different programming languages. Some controllers do require additional features such as USB connectivity, or GPIO pins. Finally, it shall be noted, that all API calls are idempotent.

4.2.2 Dimmer

The Dimmer device consists of three hardware parts. (a) A Raspberry Pi 2 provides Ethernet connectivity, exposes the functionality on a SocketIO API, and serves the WebUI. Since the zero-crossing power control presented in Section 3.4.2 requires real-time capabilities beyond what the Raspberry Pi can achieve on its own, it connects to (b) an Ardunio via a USB Serial connection. The Arduino, being microcontroller-based, allows for a more responsive control of the digital outputs. It also exposes interrupt lanes on its GPIO interface perfectly suitable to be used with a zero-crossing detector IC. The Arduino's GPIO interface is connected to (c) a custom made printed circuit board (PCB) bearing the solid-state-relays to control the power outlets, as well as the zero-crossing detection IC providing the input for the power control logic. It additionally includes all the necessary circuitry to drive the solid-state-relays using the Arduino's GPIO pins. The schematics of the PCB are available in Figure 4.5.

The REST API, being mainly intended for UI initialization, provides two calls, being (a) `/config/` exposing the Dimmer's persistent configuration and (b) `/environment/` returning

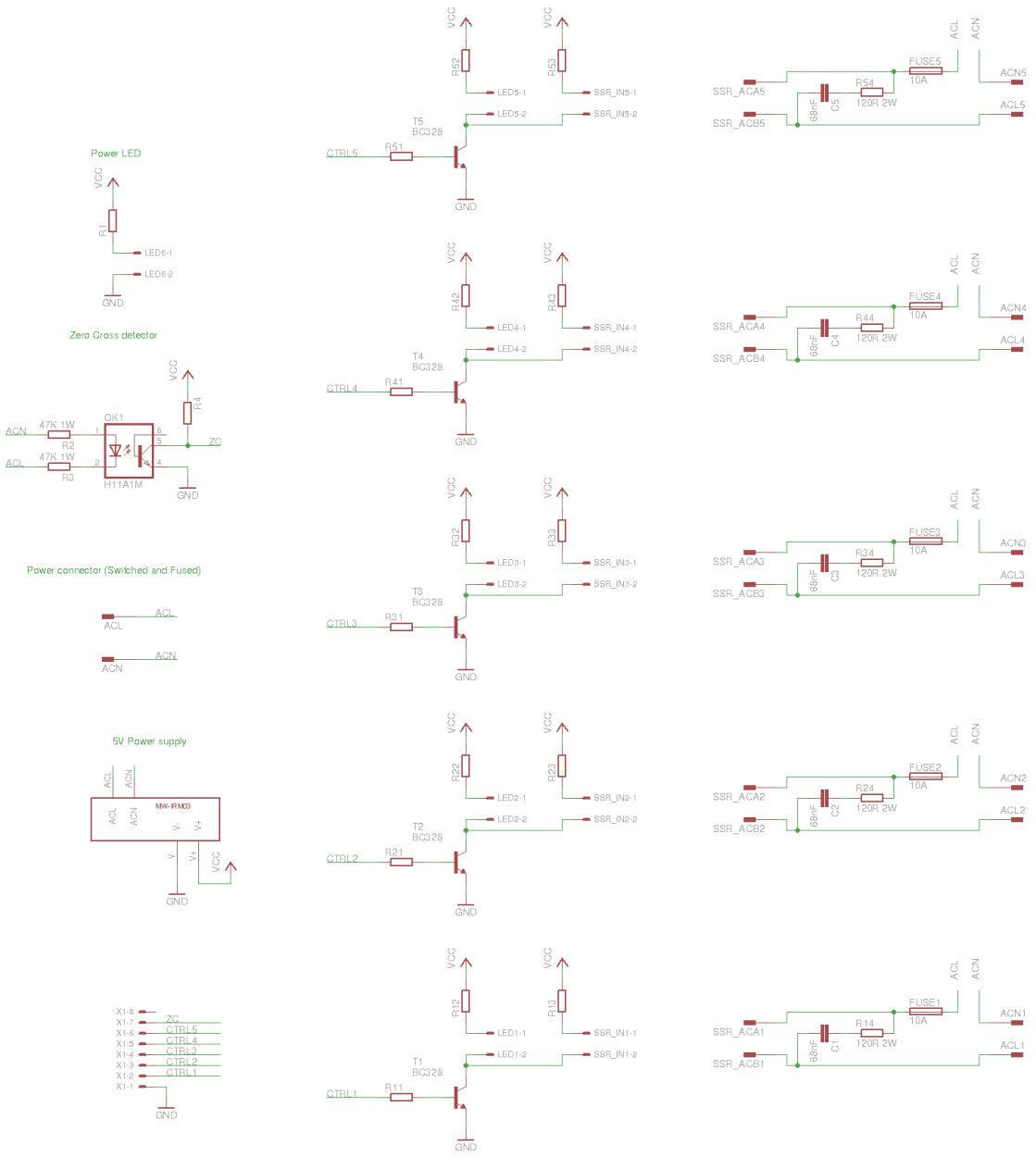


Figure 4.5: Schematic of the Dimmer's custom PCB. By courtesy of Florin Hillebrand.

Figure 4.6: The Dimmer device. The top bears the Arduino Nano and the power inlet (left), and the solid-state relays. The circuit board is located at the center. The five power outlets, as well as a mains switch are located at the bottom.

the outlet's current state. Both APIs can be queried using HTTP-GET requests and yield JSON objects of the form described below.

On its SocketIO API the Dimmer provides two SocketIO namespaces, namely (a) `events` providing control over the various power outlets and (b) `config` providing control over some configuration parameters. A state applied using the `events` namespace is ephemeral and lost on reboot, while a state applied using the `config` namespace is persistent and restored after reboot.

We shall first look at the API exposed via the REST API, followed by the one provided by the SocketIO server. Finally, we will see how the Raspberry Pi and the Arduino communicate with each other over the USB Serial connection.

JSON objects

The current state of the Dimmer's outlets is represented by a JSON object containing four properties, each containing the state for one of the outlets. The first outlet is represented by (a) `fan_level` of type Integer [0, 16] containing the currently configured power level for the first outlet. Being intended to be used by off-the-shelf 220V AC fans, the power level is the input applied to the zero-crossing power control logic presented in Section 3.4.2. (b) `watersplasher_state` is a Boolean containing the currently configured state of the second outlet. It shall be noticed, that the second outlet is a PWM outlet, hence its actual state is a superposition of this value and the configured PWM duty cycle. Whenever

this value is set to `false` the outlet is switched off, though the same can be achieved by setting `watersplasher_intensity` (see below) to 0 and `watersplasher_state` to any of `true` or `false`. Finally, the third and fourth outlet, being configured as simple on/off controls, are represented by the Booleans (c) `heat` and (d) `cold` respectively. A value of `true` means the power outlet is on, while a value of `false` means the power outlet is off. An example of such a JSON object can be found in Listing 4.1.

Listing 4.1 Dimmer's current ephemeral state

```
{
    "cold":false,
    "fan_level":0,
    "heat":false,
    "watersplasher_state":false
}
```

The current configuration of the Dimmer is represented by a JSON object containing a single property `watersplasher_intensity` of type Float [0.0, 1.0]. It contains the currently configured PWM duty cycle for the second outlet. An example for such a JSON object can be found in Listing 4.2.

Listing 4.2 Dimmer's current persistent configuration

```
{
    "watersplasher_intensity":0.25
}
```

REST API

The Dimmer provides a REST API exposing both its ephemeral state, and the persistent configuration. This API may be used by all clients, though its main purpose is UI initialization.

- `/environment/`
  - *Method:* GET
  - *Parameters:* None
  - *Returns:* The current state of the outlets. See Listing 4.1 for an example.
- `/config/`
  - *Method:* GET
  - *Parameters:* None
  - *Returns:* The persistent configuration. See Listing 4.2 for an example.

Namespace `events`

The `events` namespace provides control over the devices attached to the Dimmer's 220V AC outlets. The following events are supported:

- `update`
  - *Source:* Server
  - *Payload:* JSON containing the current outlet state of the device. See Listing 4.1 for an example.
  - *Function:* Notifies clients about outlet state changes.
- `unityFanSpeedEvent`
  - *Source:* Client
  - *Payload:* Integer [0, 16]
  - *Function:* Sets the zero-crossing based PWM duty cycle (see Section 3.4.2 for further details) for the first outlet from off (0) to full-on (16) with 15 intermediate steps.
- `unityWaterSplasherEvent`
  - *Source:* Client
  - *Payload:* Integer [0, 1]
  - *Function:* Sets the PWM value of the second outlet to either the value configured using `config/waterSplasherDutyCycle` (1) or switches the outlet off (0). The power state is toggled once a second, with the PWM value configured using `config/waterSplasherDutyCycle` defining the duty cycle. Note, that this method does not respect zero-crossings on the 220 V AC input.
- `unityHeatEvent` / `unityColdEvent`
  - *Source:* Client
  - *Payload:* Integer [0, 1]
  - *Function:* Switches the third/fourth outlet either on (1) or off (0). Notice, that this method does not respect zero-crossings on the 220 V AC input.

Namespace `config`

The `config` namespace provides means for setting up the Dimmer and its connected devices. These calls are only meant to be used by interactive UIs, such as the WebUI. The following events are supported:

- `update`
  - *Source:* Server
  - *Payload:* JSON containing the current configuration of the device. See Listing 4.2 for an example.
  - *Function:* Notifies clients about configuration changes.
- `initSequence`
  - *Source:* UI
  - *Payload:* None
  - *Function:* Starts the initialization sequence for the watersplasher aiming to flush any residual air out of its tubing. This sequence may also be used for emptying the system before dismantling it.

- `waterSplasherDutyCycle`
  - *Source:* UI
  - *Payload:* Float, [0, 1.0]
  - *Function:* Sets the duty cycle for the second outlet.

USB protocol

As mentioned before, the Raspberry Pi communicates with the Arduino via a USB Serial connection. The protocol is byte-based. The messages are initiated by the Raspberry Pi and no response is expected from the Arduino. Each message has to adhere the following schema:

```
0xF6 0x6F 0x04 <channel> <value>
```

where `channel` identifies the outlet (`0x00` through `0x03`), and `value` defines the zero-crossing based PWM duty cycle with `0x00` being off and `0x10` being full-on. A non zero-crossing respecting on/off switching can be implemented by only using the values `0x00` and `0x10`.

4.2.3 EPOS controller

Similar to the Dimmer, the EPOS controller device consists of three parts, too. While (a) the Raspberry Pi is used for the same purpose as in the Dimmer device, namely providing Ethernet connectivity, exposing the functionality on a SocketIO API, and serving the WebUI, the (b) Arduino connected to it via a USB Serial connection is used mainly because of its analog-to-digital converter (ADC) component. Would the Raspberry Pi, or any device replacing it in an alternative implementation provide an ADC, the Arduino could be removed from the setup without loosing functionality. The third component is a (c) Maxon Motors EPOS2 24/5 servo motor controller, referred to as EPOS2. The EPOS2 is connected to the Raspberry Pi via USB and allows a wide range of Maxon Motors servo motors to be controlled using either USB or CAN via a binary C library provided by the manufacturer for a wide variety of platforms, including the Raspberry Pi.

For safety reasons, we put a clutch in between the servo and the mechanics tied to the user. This clutch's power connection is controlled by two limit switches which are opened in case the servo moves out of its defined operation range. Due to the servo not being directly connected to the mechanics, the position feedback from the servo cannot be used. Therefore, we measure the servo's position on the clutch using a linear 50 kΩ potentiometer. The value of the potentiometer is read using the Arduino's analog-to-digital converter on analog pin 7 and then transmitted to the Raspberry Pi via USB Serial.

In consequence, we cannot use the servo's positional mode as the servo's internal position may become offset from the mechanics position due to the clutch being open while the servo is moving. Hence, we drive the servo without regarding its positional feedback only

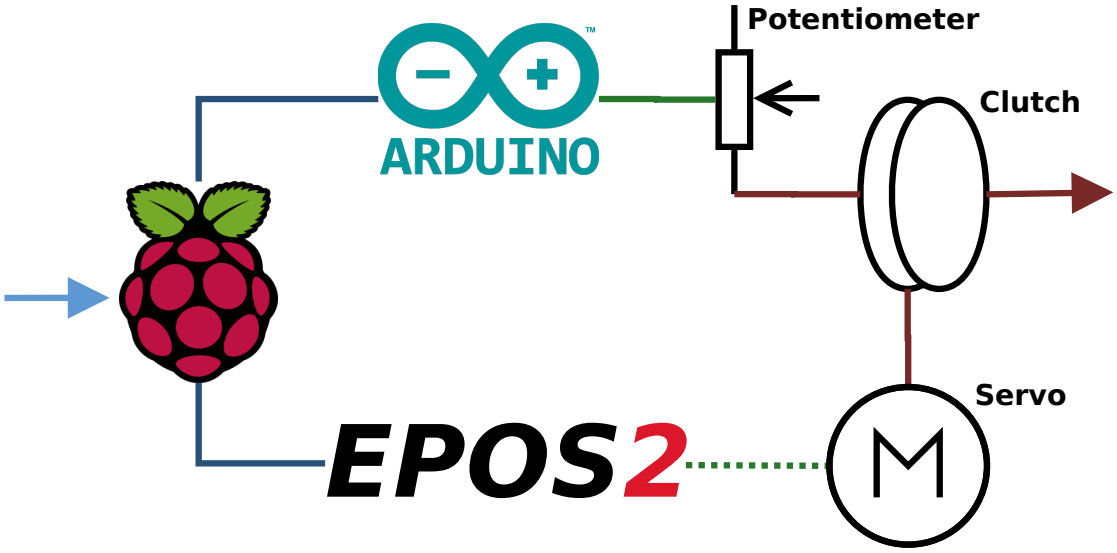


Figure 4.7: Schematic of the servo setup. Inputs are transmitted to the Raspberry Pi via Ethernet (light blue arrow). Dark blue lines represent USB connections. Solid green lines represent electrical connections. Red line represent mechanical connections. The green dotted line is a proprietary cable by Maxon Motors with unknown parameters. The ropes connected to the user are attached to the clutch (red arrow on the right hand side).

relying on the value provided by the potentiometer via the Arduino. For an overview of the full setup see Figure 4.7.

Differently from the Dimmer, the EPOS controller's REST API exposing the controller's current state is not only intended for UI initialization, but it shall be periodically queried by any connected UI. We recommend to query it twice a second as a compromise between being able to show up-to-date data and keeping the load on the API low. The reason for the necessity of exposing the current state via REST rather then emitting an event on every state change (as done by the Dimmer device) lies in the fact, that the servo's current position reported by the potentiometer may change every single time it is queried. This may by caused by electromagnetic interference on the electrical connection between potentiometer and Arduino, tolerance of both the Arduino's ADC and the potentiometer, or simply because the servo is actually moving. For the internal logic we want to have as up-to-date data as possible, which means the potentiometer's value is queried and transmitted to the Raspberry Pi as often as possible. Hence, emitting an event on every single position change will inevitably flood the network. In addition, it will not provide any benefit to the user as the values change faster then a user may react to them.

On its SocketIO API the EPOS controller provides two SocketIO namespaces, namely (a) `servo` providing control over the servo's target state, and (b) `config` providing the ability to recalibrate the servo in case it becomes offset because of friction between the clutch and the potentiometer, or during transport. Differently from the Dimmer, all state

Figure 4.8: The EPOS controller device. The left-hand side image shows the controller, with the Arduino Nano (top left), and the Maxon Motors EPOS2 24/5 Digital positioning controller and the Raspberry Pi (bottom right). The blue cylinder at the top right is a capacitor needed to smooth out voltage spikes coming from the power supply. The right-hand side image depicts the controller including (from left to right) the clutch and the servo motor.

applied via both the EPOS controller's `servo` and `config` namespaces are ephemeral and not persisted on reboots.

JSON objects

The exposed internal state of the EPOS controller is represented by a JSON object containing information about both the current state and the target state of the servo. In total we have five properties, each representing a different aspect of one of the states. The most basic properties are (a) `enabled`, a Boolean telling whether (`true`) or not (`false`) the servo motor is activated (i.e. allowed to move) and (b) `move_state`, an Integer [0, 2] giving additional information about the current moving direction. The possible values are `0` (servo is stopped), `1` (servo is moving towards positions with higher values), and `2` (servo is moving towards positions with lower values). The current and target positions are represented by (c) `current_poti_position` and (d) `target_position`, both of type Integer [0, 1023]. It shall be noted, that `current_poti_position` contains the potentiometer value as returned by the Arduino without the calibration offset applied. Finally, (e) `current_offset` is an Integer containing the current recalibration offset. It shall be noted, that the EPOS controller aims to reach equilibrium by making `current_poti_position` the same as `target_position - current_offset`. An example of such a JSON object can be found in Listing 4.3.

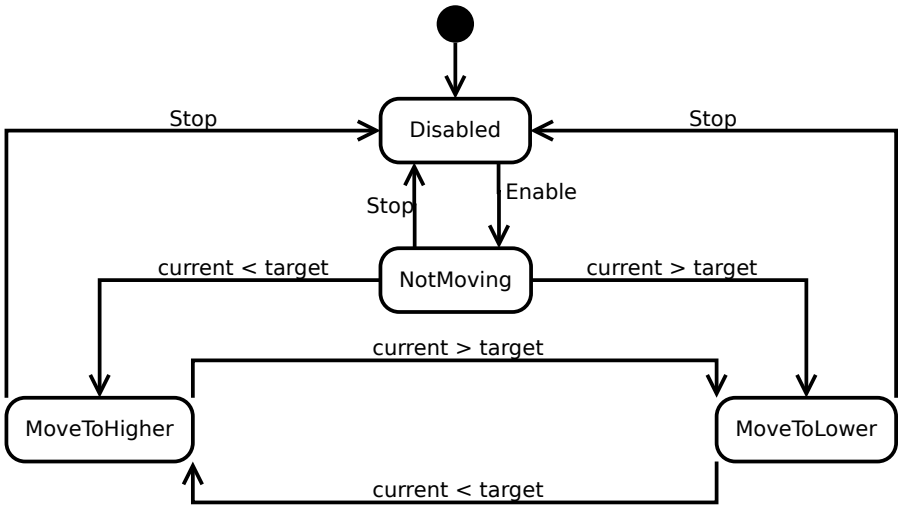


Figure 4.9: The EPOS controller state diagram.

Listing 4.3 EPOS controller's current state

```
{
    "current_offset":0,
    "current_poti_position":600,
    "enabled":false,
    "move_state":0,
    "target_position":512
}
```

REST API

The EPOS controller provides a REST API exposing the state of the servo motor setup. This API may be used by all clients, though its main purpose is UI initialization.

- `/status`
  - *Method:* GET
  - *Parameters:* None
  - *Returns:* The current internal state. See Listing 4.3 for an example.

Namespace `config`

The `config` namespace provides means for re-calibrating the servo.

- `resetCenter`
  - *Source:* UI
  - *Payload:* None
  - *Function:* Tells the server, that the servo's current position is the new center position.

Namespace `servo`

The `servo` namespace provides control over the servo's state.

- `enable`
  - *Source:* Client
  - *Payload:* None
  - *Function:* Enables the servo. The servo only moves if it is enabled. The internal state of the controller (like the target position applied via `moveTo`) may be updated in both enabled and disabled mode.
- `moveTo`
  - *Source:* Client
  - *Payload:* Integer [0, 1023]
  - *Function:* Sets the target position for the servo. Position 512 is center.
- `stop`
  - *Source:* Client
  - *Payload:* None
  - *Function:* Immediately stops the servo motor and puts it in disabled mode. `enable` has to be invoked for the servo to start moving again. The internal state (e.g. target position) is preserved. If a client emits the `enable` at any point in time it shall `stop` the servo before disconnecting.

USB protocol

Similar to the Dimmer, for this aspect too the Arduino and the Raspberry Pi communicate with each other via a USB Serial connection. The messages are initiated by the Arduino and no response is expected from the Raspberry Pi. The protocol is ASCII-based. Each message has to adhere the following schema:

$$\#\texttt{<position> <ok>}\backslash n$$

where `position` contains the clutch's current position in the range [0, 1023]. The values match range and orientation of those to be provided to the `servo/moveTo` event. `ok` contains a single Integer in the range [0, 1] to tell whether (1) or not (0) all of the clutch's parameters are within the specification. This value has to be present, but is ignored

by the current implementation as we opted to implement the safety features via limit switches directly controlling the power supply of the clutch. The value is present in case it will be used by later iterations of the system.

EPOS2 protocol

The Maxon Motors EPOS2 24/5 Digital positioning controller device is integrated with the system using a C-library provided by Maxon Motors. The communication protocol is USB based, proprietary, and all further details are unknown to the author.

4.2.4 Vragrancer

The Vragrancer device is the most basic of the three devices because it only consists of two devices, being (a) a Raspberry Pi 2 providing Ethernet connectivity, exposing the functionality on a SocketIO API and serving the WebUI, and (b) a custom circuit board attached to the Raspberry Pi's GPIO interface. We opted for this approach, as the main purpose is to control seven pneumatic valves on the scent dispenser. This task does neither bear the need for real-time capabilities beyond of what the Raspberry Pi is able to deliver, nor to interface with a third-party device using a complex communication protocol, such as Serial or USB. Controlling the power states of each of the seven valves is sufficient.

The actual scents are contained in a single cartridge, containing six different scents. The cartridge may be swapped for the individual simulations. Hence, it is not possible to know a priori which of the six slots contains which scent. For this reason we decided to use generic numeric IDs to refer to each of the six slots, rather then to name the slots by the scent they respectively contain.

The Vragrancer provides a single namespace on its SocketIO API, being `scent`. It provides control over the various scents and notifies the UIs about state changes. State applied using the `scent` namespace is ephemeral and lost on reboot.

It shall be noted, that due to its simplicity, the Vragrancer does not provide a REST API as the other devices do.

JSON objects

The exposed internal state of the EPOS controller is represented by a JSON object containing a property for each of the six scents it is able to control. The property name is a String representation of its index, starting with 0, referred to as the scent's ID. The value of each of the properties is a Boolean telling whether the scent is currently active (`true`) or not (`false`). An example of such a JSON object can be found in Listing 4.4.

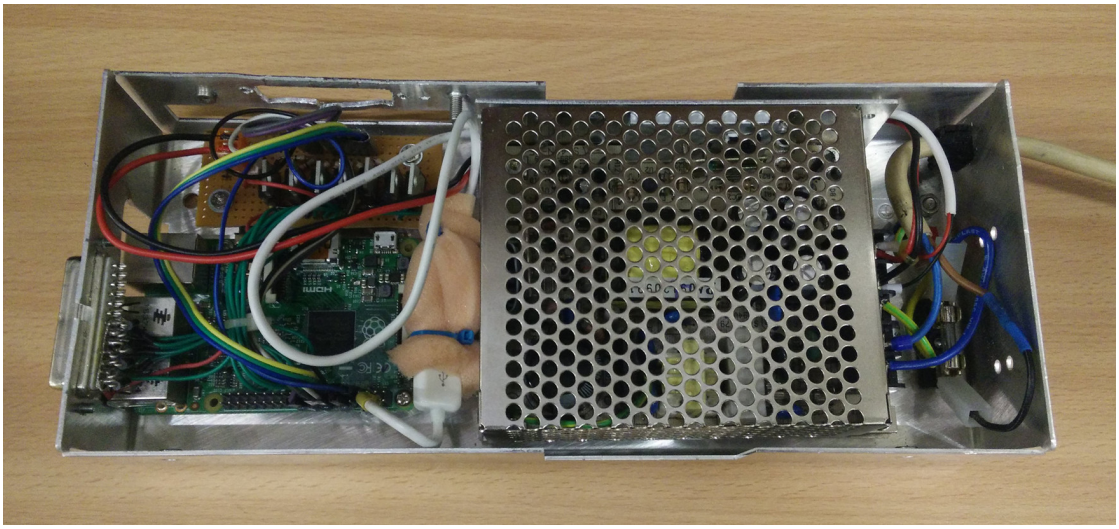


Figure 4.10: The Vragrancer device. Right-hand side is the 12 V power supply. On the bottom left is the Raspberry Pi, top left bears the transistor circuitry.

Listing 4.4 Vragrancer's current state

```
{
    "0": false,
    "1": false,
    "2": true,
    "3": false,
    "4": true,
    "5": false
}
```

Namespace `scent`

The `scent` namespace provides control over the Raspberry Pi's GPIO pins controlling the scents. Any combination of scents can be active at any point in time. After being activated, a scent stays active for one second and is then automatically deactivated. If the scent shall be active for longer, multiple calls to `activate` have to be invoked. This extends the activity period of a scent to one second after the last `activate` call was received, assuming none of the deactivation calls were issued.

- `activate` / `deactivate`
  - *Source:* Client
  - *Payload:* Integer [0, 5] or String representation thereof
  - *Function:* Activate/deactivate the scent with the given ID.
- `deactivateAll`
  - *Source:* Client
  - *Payload:* None
  - *Function:* Deactivate all scents.
- `status_changed`
  - *Source:* Server
  - *Payload:* JSON containing the new state. See Listing 4.4 for an example.
  - *Function:* Notifies all connected clients about a change of the internal state. In particular, `status_changed` is emitted whenever a client connects. This event is mainly intended for visual consistency over multiple UIs.

4.3 Control terminal

The control terminal aims to provide the user with a fully integrated UI to control all parts of the system. For this, the control terminal includes all the WebUIs of the individual devices in a single webpage using HTML inline frames (`<iframe>`). The advantage of this approach is, that the single WebUIs can be updated individually by updating the respective device and the control terminal does not need to be updated every time a single device is. Furthermore, this approach allows to bypass modern browser's same-origin policy. Same as the WebUIs, the control terminal can be used both with mouse and keyboard, as well as with a touchscreen. Figures 4.11 through 4.13 illustrate the web interfaces of the various system components.

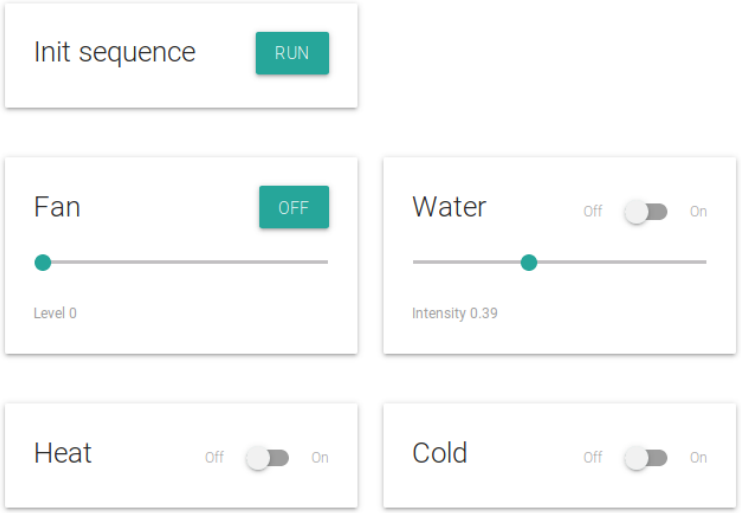


Figure 4.11: WebUI of the Dimmer. *Init sequence* allows to trigger the watersplasher's initialization sequence (event `config/initSequence`). *Fan* allows to control the first outlet (event `events/unityFanSpeedEvent`). *Water* allows to control the second outlet (event `events/unityWaterSplasherEvent` using the switch and event `config/waterSplasherDutyCycle` using the slider). *Heat* allows to control the third outlet (event `events/unityHeatEvent`). *Cold* allows to control the fourth outlet (event `events/unityColdEvent`).

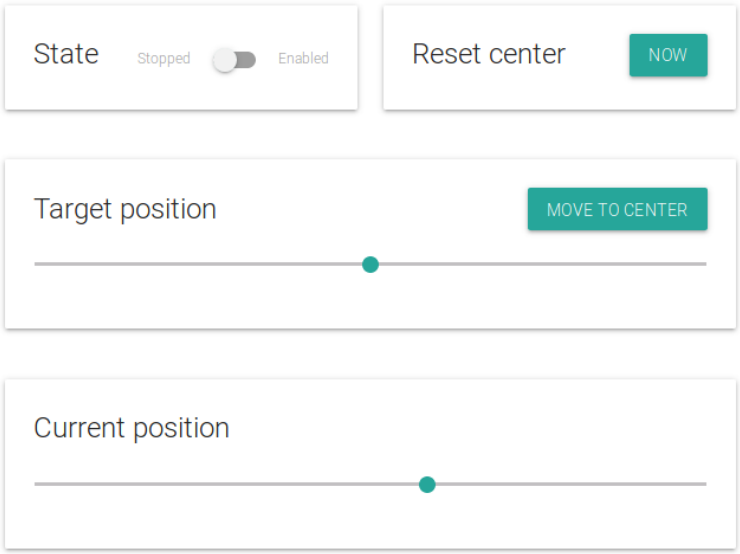


Figure 4.12: WebUI of the EPOS controller. *State* allows to enable and disable the servo (events `servo/enable` and `servo/stop`). *Target position* allows to set the target position (event `servo/moveTo`). *Current position* gives visual feedback about the servo's current position. *Reset center* allows to recalibrate the servo (event `config/resetCenter`).

Figure 4.13: WebUI of the Vragrancer. The switches allow to control the individual scents (events `scent/activate` and `scent/deactivate`). The *Stop all scents* button allows to switch off all cents at once (event `scent/deactivateAll`).

CHAPTER 5

Feedback and discussion

In this chapter we analyze how the system performs within the context of Virtual Reality. For that, we evaluate data gathered from users of the system, which includes both a quantitative approach using physiological data, and a qualitative one using questionnaires.

5.1 Presence

As we have seen, the multi-sensory system is able to meet all functional requirements imposed upon it. Before we can analyze how additional sensory stimuli affect the user experience, we need to show that even in its most rudimentary setup, the Jumpcube is able to deliver a sense of presence.

5.1.1 Testing methodology

To determine if the Jumpcube delivers a sense of presence we conducted an experiment involving both an experienced skydiver (Stefanie Liller) and a member of the project (Jonas Röthlin), who has never experienced a skydive in real life. We opted for a physiological measurement, which even though being more effort then a questionnaire-based approach, was regarded by the project team to be feasible with such a low number of participants and to yield more objective data.

We used an early version of the *Vienna Parachute Jump* simulation, only featuring the wind simulation. It shall also be noted that this simulation does not offer any user interaction. This in turn should cause the measured physiological responses to be mostly caused by psychological effects, hence a measure of presence can be derived from it. The data was collected using a heart rate measurement unit attached to the user's chest.

To better understand the gathered data, one needs to know that the simulation consists of four phases with different characteristics. Due to these differences we assumed them to cause different physical reactions on the test subjects. The four phases are:

1. *Pre-jump:* The user stands in the virtual air plane at high altitude and waits for the signal to jump
2. *Skydive:* The user falls towards the city of Vienna in a simulation of a tandem skydive
3. *Parachute glide:* After the parachute opens the user glides over the city of Vienna
4. *Landing:* After flying through its roof, the user lands in the Kuppelsaal of Technische Universität Wien's main building.

To easily distinguish these phases we also collected acceleration values using a regular smartphone with built-in accelerometer. Each transition should cause a noticeable spike in the measured acceleration which allows us to relate heart rate and phase.

5.1.2 Heart rates

As can be seen from Figure 5.1, we found the heart rate fluctuating on both test subjects, spiking well over 100 beats-per-minute when leaving the plane. On the skilled skydiver we see a second spike above 100 beats-per-minute from sample 160 to sample 180. This was later attributed by the test subject to some virtual airships passing right below her. As she pointed out, having something right below one while skydiving is highly dangerous, and the measurement clearly shows a physical reaction. Another spike can be seen on both subjects at the very end of the simulation, which is the moment the test subject approaches Technische Universität Wien's main building and flies through the roof of the Kuppelsaal.

5.1.3 Conclusion

Overall, we can deduct, that the Jumpcube does provide a feeling of presence, both to users being familiar with the simulated experience, as well as with those who are not. The experienced skydiver showed a physiological reaction when the airships passed right below her, a reaction not shown by the other user. We attribute this to her recognizing it as a dangerous situation, while the other subject, due to his lack of experience, did not.

5.2 Impact of the multi-sensory system

As we have seen, we have reason to assume the Jumpcube in its original state was already able to deliver a feeling of presence to the user. We will now see how the additional multi-sensory stimuli did affect the user's experience.

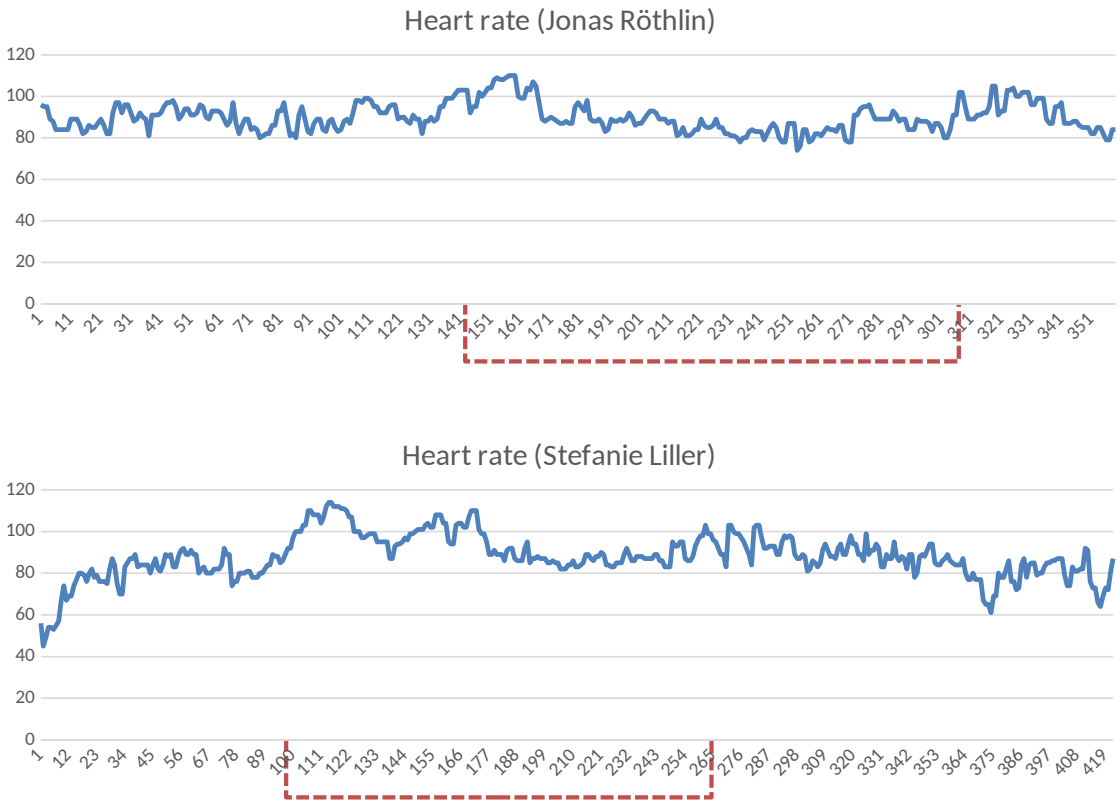


Figure 5.1: Mesured heart rates of the unexperienced test subject (Jonas Röthlin) and the experienced skydiver (Stefanie Liller) over time (in seconds). The red lines on either plot show the duration of the simulation. The time before refers to jump preparation.

5.2.1 Testing methodology

We conducted an anonymous questionnaire-based survey of the user experience during the Jumpcube's various public and semi-public appearances in 2017 and during its appearance at the European Congress of Radiology (ECR) in 2018. The full questionnaires can be found in Appendix A of this thesis. The participants filled out the questionnaires on a voluntary basis, and they were not given any additional instructions before participating.

The participants were free to choose from the following simulations:

- *Skydive:* A non-interactive skydive from high altitude above the city of Vienna.
- *Mars:* An interactive flight through the solar system.
- *Airrace:* An interactive race against a plane through a canyon.
- *Diving:* An underwater experience with very little interactivity (only ECR 2018).
- *Vienna Airrace:* An interactive race against a plane, partly at low altitude over the city of Vienna, partly through the sewers thereof (only ECR 2018).

The simulations feature the stimuli listed in below overview. An x denotes the presence of a stimulus. The last column lists the interactivity level of the simulation, 0 being non-interactive and 3 being highly interactive.

Simulation	Wind	Force	Odors	Moisture	Interactive
Skydive	x		x	x	0
Mars	x	x	x		2
Airrace	x	x	x	x	3
Diving			x	x	1
Vienna Airrace	x	x	x	x	3

Finally, it shall be noted, that between the questionnaires the centrifugal force component of the multi-sensory system has been reworked to deliver stronger stimuli.

5.2.2 Questionnaire analysis

Overall, we gathered 197 questionnaires during the various public and semi-public appearances in 2017, and 103 questionnaires during the ECR 2018, the Jumpcube's only appearance in 2018.

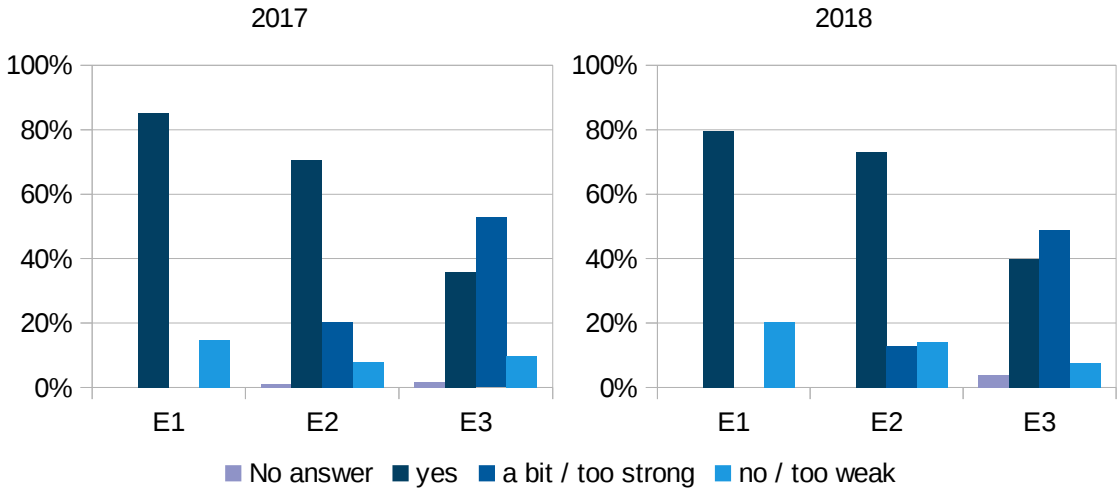
From question *Was the physical jump exciting for you?*, having very similar results in both 2017 (yes/mostly/no = 75.5%/20.9%/3.6%) and 2018 (76.2%/21.8%/2.0%) we can already see, that augmenting Virtual Reality with real world components can improve the user experience of Virtual Reality applications.

Moving to the multi-sensory feedback, we can see that a high number of participants noticed the smells (Question A1 in 2017/2018 = 85.2%/76.5%). Of those, most found them to be adequate (Question A3 in 2017/ Question A2 in 2018 = 70.7%/73.1%) and attributed them to make the experience more interesting at least to some degree (Question A4 in 2017/ Question A3 in 2018 = 90.3%/88.5%). The results are plotted in Figure 5.2.

We can also see a clear impact of our late-2017 rework on the centrifugal force component, as the percentage of people perceiving them rose from 58.9% in 2017 to 80.6% in 2018 (Question B4 in 2017 / Question C2 in 2018). However, it does not seem that they were perceived as more realistic by those noticing in 2018 (Question B5 in 2017 / Question C3 in 2018; yes/mostly/no = 33.3%/57.4%/7.4%) compared to 2017 (yes/mostly/no = 32.6%/62.8%/4.7%). The results are plotted in Figure 5.3.

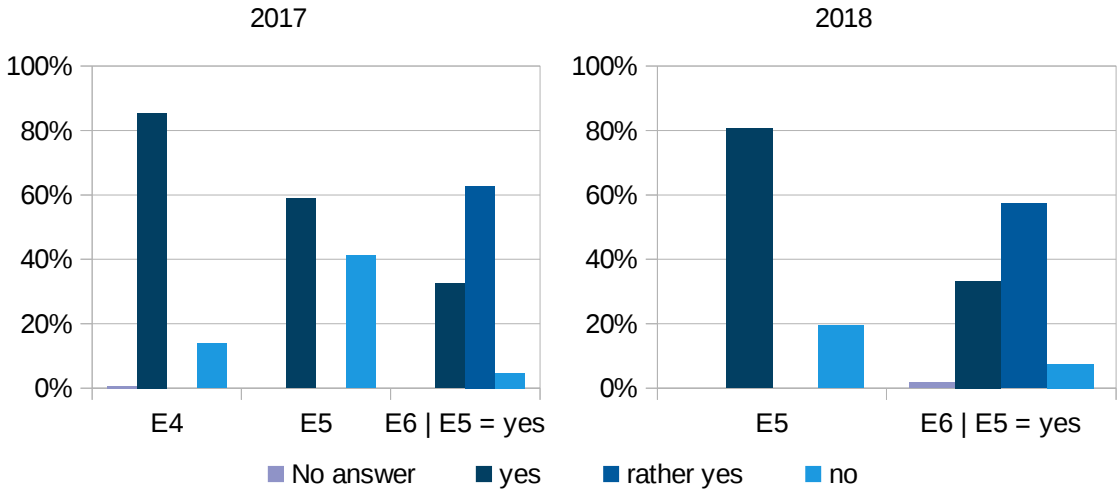
We found in 2017, that moisture (Question B2 in 2017) was perceived by 86.0% of the participants, the highest value of all analyzed sensory stimuli. Since we did not change the component significantly, we decided to drop the question in 2018 for a more relevant one. The plot herefor can be found in Figure 5.3.

Overall we found high acceptance by the participants. Almost all participants found the Jumpcube's simulations to be at least "exciting" with the vast majority certifying "very exciting" in both 2017 (Question C4, very/a little/no = 79.9%/18.1%/2.1%) and 2018 (Question D4, very/a little/no = 88.9%/11.1%/0.0%). Though, it shall be noted,



E1: Did you perceive smells in the virtual reality experience?
E2: If 1. yes, how intense were the smells?
E3: If 1. yes, did the smells make the virtual reality experience more interesting?

Figure 5.2: Users' reactions to olfactory stimuli.



E4: Did you notice the humidity of the clouds?
E5: Did you recognize the centrifugal forces in curves?
E6: If yes, were the centrifugal forces realistic?

Figure 5.3: Users' reactions to haptic stimuli.

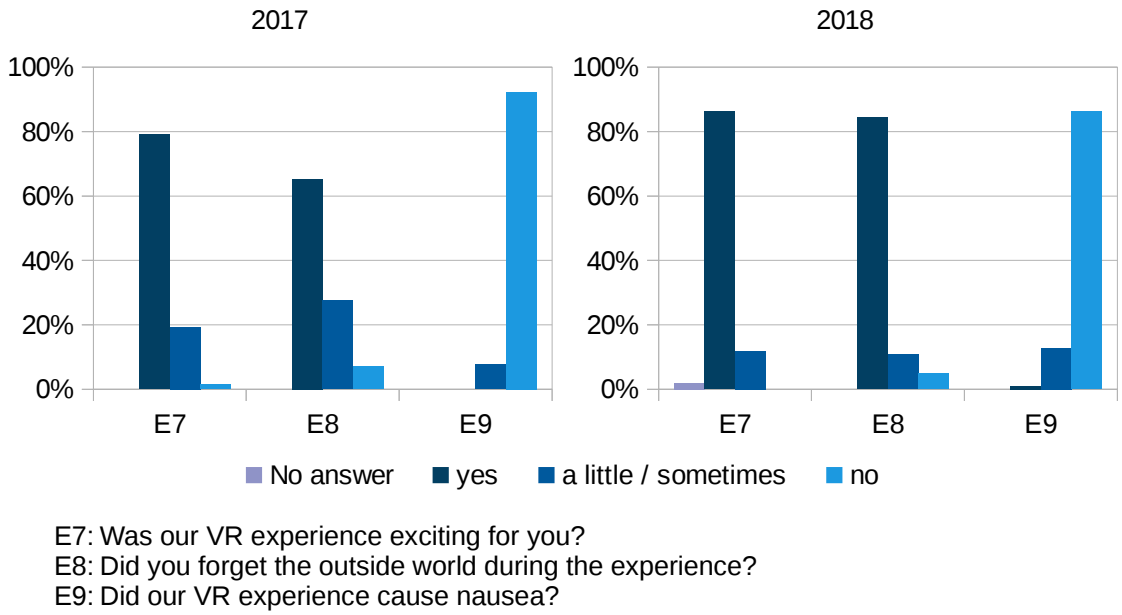


Figure 5.4: Users' overall impressions.

that when including only the simulations which offer support for centrifugal forces, we see the 2018 values (very/a little/no = 81.3%/18.8%/0.0%) being similar to the 2017 ones. Hence, the increase in excitement cannot be attributed to the centrifugal force component rework.

Additionally, we see that the Jumpcube users experienced very little to no simulator sickness in both 2017 (Question C6, yes/a little/no = 0.0%/7.7%/92.3%) and 2018 (Question D6, yes/a little/no = 1.0%/12.9%/86.1%). For the Airrace the reported nausea-per-jump ratio is highest (2017/2018 = 16.6%/21.7%). This was expected by the team, as Airrace is both the most demanding simulation from a physical perspective, and due to its high interactivity bears the highest risk of disorientation and the possibility to perform actions in the virtual world, which would lead to serious injury and death if performed equally in the real world.

Finally, we see a significant increase in perceived presence from 2017 (Question C5, yes / sometimes / no = 65.3%/27.6%/7.1%) to 2018 (Question D5, yes / sometimes / no = 84.3%/10.8%/4.9%). We can also see a slight improvement when directly comparing only the simulations which offer centrifugal forces 2017 (yes / sometimes / no = 63.4%/30.5%/6.1%) to the same simulations 2018 (yes / sometimes / no = 75.0%/15.6%/9.4%). This leads us to the assumption, that the centrifugal force component rework did have a positive influence on perceived presence, but it alone does not explain the increase. A plot of the results for these general questions can be found in Figure 5.4.

In summary, the questionnaires have shown, that multi-sensory feedback is both noticed and appreciated by the majority of the users. From this we conclude that the system is

stable and user-friendly in application. We have also seen, that the Jumpcube is able to deliver a feeling of presence to the user, without causing simulator sickness. We can thus conclude, that the addition of a multi-sensory feedback system benefits the Virtual Reality user experience delivered by the Jumpcube.

CHAPTER 6

Conclusion and outlook

In the final chapter, first we give a short summary over the contents of this thesis, followed by a few suggestions for future work and improvements, both from a technical perspective and on further user studies which may be conducted using the Jumpcube's multi-sensory system.

We have seen how Virtual Reality has evolved from the beginnings in the late 1960s with the work of Virtual Reality pioneer Morton Heilig to its current state. We have analyzed both the technical basis of the technology, as well as the features of human physiology and psychology, which allow for the technology to create a sense of presence. We have also seen how Virtual Reality used technological advancements to increase immersion, while at the same time lowering costs. Our conclusion here is that while still facing some challenges, Virtual Reality in its current form is able to simulate situations with a degree of fidelity high enough to be used for both simple entertainment, and therapeutic purposes.

From a technological point of view we have seen, how game engines are of significant importance when developing for Virtual Reality. We have also seen, how open source software and hardware help to deal with the challenges faced during the development of an interactive hardware project. We determined that modern scripting languages such as Python and web standards such as WebSockets are in the present case best suited for rapid prototyping, as they are easily integrated with existing software and hardware. The accompanying performance drawbacks proved to be insignificant for the use in Virtual Reality because available components such as the Raspberry Pi 2, or 100MBit Ethernet provide enough resources to compensate them. In addition, these parts are easily available on the free market at a reasonable price. Finally, we have seen how we can automate the build process using the Yocto project to easily build ready-to-use SD card images in a reproducible way.

From our experiments, we can see, that the Jumpcube as a whole provides a high degree of both presence and immersion. This has been shown by a physiological measurement, which showed significant responses on both the experienced skydiver, as well as the test candidate, who never experienced comparable situations in real life. Further on, our questionnaire series shows, that the vast majority of the users both notice multi-sensory stimuli, and attribute them to be an improvement in the overall user experience.

However, the project still offers room for further improvements, both from the technological point of view, as well as from the scientific one. For the sooner, a first improvement would be to implement an automated discovery mechanism for the single controller devices. Currently, the connection establishment relies on static IP addresses defined at build time. It shall be noted that we regard this is an appropriate solution for standalone entities such as the Jumpcube, as it does not add to the complexity of the system at runtime. But, we presume that this approach will scale badly with the addition of more controllers. Additionally, it enforces the use of a 192.168.1.0/24 network. Similarly to the discovery mechanism, this does not limit the Jumpcube in any way, though it may affect how the system can be used in other environments. Finally, it shall be noted that we did not address the issue of having to update the controllers' operating system and application. Currently, we need to swap SD cards to achieve this, which requires physical access to the devices. This too is presumed to scale badly with the number of controllers. However, we feel confident that the hardware and software stack we chose is able to cope with all of this challenges with little to no changes required to the existing components.

From a scientific perspective we have to acknowledge that our results are not representative. For our physiological test series this can be mostly attributed to the low number of participants, which in turn is caused by the complexity of the method. To improve on this we need a larger, representative test sample. For that we need to define a standardized test procedure and means of automating the evaluation of the gathered data. Without such a procedure we see no possibility of conducting such an experiment efficiently. For our questionnaire based study we also identified some problems. As mentioned, the study was conducted during public or semi-public appearances of the Jumpcube. This means, that we were able to only partly control the environment. Additionally, we assume that on such events a high percentage of the test subjects share similar demographics, and state of mind. On the other hand, the events used to conduct the user study were highly different between each other, ranging from renowned symposia such as the European Congress of Radiology, to pure public relations appearances, such as one in the Wien Museum. This leads us to the assumption, that in fact we did cover different user types, though we believe that there are not enough of in order to proclaim the study as representative.

Finally, because on each of the Jumpcube shows we wanted to provide each user with the best experience possible, we were not able to gather isolated data which would allow us to analyze which impact a single feature has on the user experience. This could be achieved in a future study by letting one user group test the simulation with said feature enabled, while a control group would experience the same simulation with said feature disabled.

List of Figures

List of Tables

Listings

Bibliography

[Abr14] Micheal Abrash. What vr could, should, and almost certainly will be within two years, 2014.

[Ala14] Mohammed M Alani. Tcp/ip model. In *Guide to OSI and TCP/IP models*, pages 19–50. Springer, 2014.

[AMDa] AMD.

[AMDb] AMD. Amd socket am4 platform.

[Ard] Arduino. Web client.

[Arm] Arm. Cortex-m series.

[ARS97] Peter J. McNerney Ernie Eastlund Brian Manson Jon Gratch Randy Hill Albert Rizzo, Jarrell Pair and Bill Swartout. Development of a vr therapy application for iraq war veterans with ptsd, 1997.

[Aya04] Kenneth J Ayala. *The 8051 microcontroller*. Cengage Learning, 2004.

[BaKI] Body Brain and Self Laboratory (Group Ehrsson) at Karolinska Institutet.

[Bar12] Steven F Barrett. Arduino microcontroller: Processing for everyone! *Synthesis Lectures on Digital Circuits and Systems*, 7(2):1–371, 2012.

[BBP+08] F. Bonato, A. Bubka, S. Palmisano, D. Phillip, and G. Moreno. Vection change exacerbates simulator sickness in virtual environments. *Presence*, 17(3):283–292, June 2008.

[BC98] Matthew Botvinick and Jonathan Cohen. Rubber hands 'feel' touch that eyes see. *Science*, 391:756, 1998.

[Ben02] Alan J. Benson. *Chapter 35 Motion Sickness*. 2002.

[BEW+98] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz. Designing a pc game engine. *IEEE Computer Graphics and Applications*, 18(1):46–53, Jan 1998.

[BM07] D. A. Bowman and R. P. McMahan. Virtual reality: How much immersion is enough? *Computer*, 40(7):36–43, July 2007.

[BORB99] Renato Alarcon David Ready Fran Shahar Ken Graap Jarrel Pair Philip Hebert Dave Gotz Brian Wills Barbara Olasov Rothbaum, Larry Hodges and David Baltzell. Virtual reality exposure therapy for ptsd vietnam veterans: A case study. *Journal of Traumatic Stress*, 12(2), 1999.

[Bro] Holly Brockwell. Forgotten genius: the man who made a working vr machine in 1957.

[Bun] Christoph Bungert. Hmd/headset/vr-helmet comparison chart.

[Bur18] Andrew Burnes, 2018.

[CHHR17] Mike Cantelon, Marc Harter, TJ Holowaychuk, and Nathan Rajlich. *Node. js in Action*. Manning Publications, 2017.

[Cry] Crytek. Cryengine v manual, section plugin system.

[Dec05] J. D. Decotignie. Ethernet-based real-time and industrial communications. *Proceedings of the IEEE*, 93(6):1102–1117, June 2005.

[Der] Benoit Dereau.

[Des05] Ajay V Deshmukh. *Microcontrollers: theory and applications*. Tata McGraw-Hill Education, 2005.

[DIL81] Nicanor P DeMesa III and John E Laabs. Bus collision avoidance system for distributed network data processing communications system, July 28 1981. US Patent 4,281,380.

[DS00] Dietmar Dietrich and Thilo Sauter. Evolution potentials for fieldbus systems. In *Factory Communication Systems, 2000. Proceedings. 2000 IEEE International Workshop on*, page 343. IEEE, 2000.

[DS11] A. Drumea and P. Svasta. Designing low cost embedded systems with ethernet connectivity. In *2011 IEEE 17th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, pages 217–220, Oct 2011.

[Edw] Benj Edwards. Unraveling the enigma of nintendo's virtual boy, 20 years later.

[Ehr07] H. Henrik Ehrsson. The experimental induction of out-of-body experiencey. *Science*, 317:1048, 2007.

[Ell09] S. R. Ellis. Latency and user performance in virtual environments and augmented reality. In *2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pages 69–69, Oct 2009.

[EMCL94] Sheldon Ebenholtz, M M Cohen, and Barry Linder. The possible role of nystagmus in motion sickness: A hypothesis. 65:1032–5, 12 1994.

[Eng] Unreal Engine. Custom udk to ship with all oculus rift developer kits.

[FFS+02] A. Flammini, P. Ferrari, E. Sisinni, D. Marioli, and A. Taroni. Sensor interfaces: from field-bus to ethernet and internet. *Sensors and Actuators A: Physical*, 101(1):194 – 202, 2002.

[FS11] Kevin R Fall and W Richard Stevens. *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley, 2011.

[FW57] Edgar F Fincham and John Walton. The reciprocal actions of accommodation and convergence. *The Journal of physiology*, 137(3):488–508, 1957.

[Gama] Epic Games. Http-requests.

[Gamb] Epic Games. Unreal engine documentation, section blueprints.

[Gar16] Gartner. Gartner's 2016 hype cycle for emerging technologies identifies three key trends that organizations must track to gain competitive advantage, 2016.

[Gar17] Gartner. Top trends in the gartner hype cycle for emerging technologies, 2017, 2017.

[God08] Atul P Godse. *Microprocessors & microcontrollers*. Technical publications, 2008.

[Gre14] Jason Gregory. *Game engine architecture*. AK Peters/CRC Press, 2014.

[Har15] William Harrington. *Learning Raspbian*. Packt Publishing Ltd, 2015.

[Heia] Morton L Heilig. Sensorama simulator.

[Heib] Morton L Heilig. Stereoscopic-television apparatus for individual use.

[Hor] Ken Horowitz. Great idea or wishful thinking?

[HR95] Ian P Howard and Brian J Rogers. *Binocular vision and stereopsis*. Oxford University Press, USA, 1995.

[Hua17] H. Hua. Enabling focus cues in head-mounted displays. *Proceedings of the IEEE*, 105(5):805–824, May 2017.

[Hun02] Craig Hunt. *TCP/IP network administration*, volume 2. " O'Reilly Media, Inc.", 2002.

[IET89] IETF. Requirements for internet hosts – communication layers, 1989.

[IET05] IETF. Uniform resource identifier (uri): Generic syntax, 2005.

[IET11] IETF. The websocket protocol, 2011.

[IET14a] IETF. Hypertext transfer protocol – http/1.1, 2014.

[IET14b] IETF. Hypertext transfer protocol (http/1.1): Semantics and content, 2014.

[iFi] iFixit.

[Inc] Texas Instruments Inc. Miniaturized electronic circuits.

[Int10] Intel. Intel core i7-970 processor, 2010.

[JL13] H. S. Juang and K. Y. Lurrr. Design and control of a two-wheel self-balancing robot using the arduino microcontroller board. In *2013 10th IEEE International Conference on Control and Automation (ICCA)*, pages 634–639, June 2013.

[Kan] Jan Kaniewski. socketio-client-ue4.

[Kaw02] H. Kawamoto. The history of liquid-crystal displays. *Proceedings of the IEEE*, 90(4):460–500, Apr 2002.

[KGMB68] R. S. Kennedy, A. Graybiel, R. C. McDonough, and Fr. D. Beckwith. Symptomatology under storm conditions in the north atlantic in control subjects and in persons with bilateral labyrinthine defects. *Acta Oto-Laryngologica*, 66(1-6):533–540, 1968.

[KKC09] I. Kartiko, M. Kavakli, and K. Cheng. The impacts of animated-virtual actors' visual complexity and simulator sickness in virtual reality applications. In *2009 Sixth International Conference on Computer Graphics, Imaging and Visualization*, pages 147–152, Aug 2009.

[Kla04] Jeff Klaas. System-on-a-chip, November 9 2004. US Patent 6,816,750.

[Kum] Greg Kumparak. A brief history of oculus.

[LD05] Donald R. Lampton and David B. Durbin. Introduction to and review of simulator sickness research, 2005.

[LMT01] Feng-Li Lian, J. R. Moyne, and D. M. Tilbury. Performance evaluation of control networks: Ethernet, controlnet, and devicenet. *IEEE Control Systems*, 21(1):66–83, Feb 2001.

[Lov10] R. Love. *Linux Kernel Development.* Developer's Library. Pearson Education, 2010.

[Lub11] Peter Lubbers. Html5 web sockets: A quantum leap in scalability for the web. *http://www. websocket. org/quantum. html*, 2011.

[Min] Wikimedia Commons Minecraftpsyco.

[MIWB02] Michael Meehan, Brent Insko, Mary Whitton, and Frederick P. Brooks, Jr. Physiological measures of presence in stressful virtual environments. *ACM Trans. Graph.*, 21(3):645–652, July 2002.

[MRWB03] M. Meehan, S. Razzaque, M. C. Whitton, and F. P. Brooks. Effect of latency on presence in stressful virtual environments. In *IEEE Virtual Reality, 2003. Proceedings.*, pages 141–148, March 2003.

[MSK15] F. Messaoudi, G. Simon, and A. Ksentini. Dissecting games engines: The case of unity3d. In *2015 International Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6, Dec 2015.

[Mus] FM-7 Museum.

[NH81] R. N. Noyce and M. E. Hoff. A history of microprocessor development at intel. *IEEE Micro*, 1(1):8–21, Feb 1981.

[Nin] Nintendo.

[Pan] Fabio R. Panettieri". unity-socket.io-deprecated.

[PK09] Kaveh Pahlavan and Prashant Krishnamurthy. *Networking Fundamentals: wide, local and personal area communications.* John Wiley & Sons, 2009.

[Rai13] Rohit Rai. *Socket. IO Real-time Web Application Development.* Packt Publishing Ltd, 2013.

[RHK+95] Barbara Olasov Rothbaum, Larry F. Hodges, Rob Kooper, Dan Opdyke, James S. Williford, and Max North. Effectiveness of computer-generated (virtual reality) graded exposure in the treatment of acrophobia. *American Journal of Psychiatry*, 152(4):626–628, 4 1995.

[Roy] Bill Roy. socket.io-arduino-client.

[RT] Sue Marquette Poremba Ross Toro. A short history of virtual reality.

[RW12] Matt Richardson and Shawn Wallace. *Getting started with raspberry PI.* " O'Reilly Media, Inc.", 2012.

[SBH+17] Jonathan Schlueter, Holly Baiotto, Melynda Hoover, Vijay Kalivarapu, Gabriel Evans, and Eliot Winer. Best practices for cross-platform virtual reality development. In *Degraded Environments: Sensing, Processing, and Display 2017*, volume 10197, page 1019709. International Society for Optics and Photonics, 2017.

[Sch] Matthew Schnipper. Seeing is believing:the state of virtual reality.

[Sch14] Maik Schmidt. *Raspberry Pi: A Quick-Start Guide*. Pragmatic Bookshelf, 2014.

[Sev14] C. Severance. Massimo banzi: Building arduino. *Computer*, 47(1):11–12, Jan 2014.

[SK] Chaudhury S. Srivastava K, Das R C. Virtual reality applications in mental health: Challenges and perspectives. 23:83–85.

[Sla03] Mel Slater. A note on presence terminology. *Presence Connect*, 3(3), 2003.

[SLUK99] Mel Slater, Vasilis Linakis, Martin Usoh, and Rob Kooper. Immersion, presence, and performance in virtual environments: An experiment with tri-dimensional chess. 06 1999.

[Tho05] J-P Thomesse. Fieldbus technology in industrial automation. *Proceedings of the IEEE*, 93(6):1073–1101, 2005.

[Tre77] M. Treisman. Motion sickness: an evolutionary hypothesis. *Science*, 197:493–495, 1977.

[TV99] E. Tovar and F. Vasques. Real-time fieldbus communications using profibus networks. *IEEE Transactions on Industrial Electronics*, 46(6):1241–1251, Dec 1999.

[Uda09] V Udayashankara. *Microcontroller*. Tata McGraw-Hill Education, 2009.

[unia] Unity 5.4 manual, section web player.

[Unib] Unity. Scripting api.

[Unic] Unity. Www.

[Unk82] Unknown. Things to come: The pinball construction set. *Softline*, 11 1982.

[VRa] Oculus VR. 4-month unity pro trial for oculus devs.

[VRb] Oculus VR. Overview of the dk2 and sdk 0.4.

[Whe] Charles Wheatstone. On some remarkable, and hitherto unobserved, phenomena of binocular vision. 128:371 – 394.

[WS98] Bob G. Witmer and Michael J. Singer. Measuring presence in virtual environments: A presence questionnaire. *Presence: Teleoperators and Virtual Environments*, 7(3):225–240, 1998.

[WT11] K. Watanabe and S. Tachi. Verification of out of body sensations, attribution and localization by interaction with oneself. pages 111–118, March 2011.

[Zar18] Anis Zarrad. Game engine solutions. In *Simulation and Gaming*. InTech, 2018.

[Zim80] H. Zimmermann. Osi reference model - the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980.

Appendix A: Questionnaires

TU Jumpcube Questionnaire #1/2017 **English Version**

[A] Questions on the olfactory stimuli (if possible, tick the most adequate answer)

1.	Did you perceive smells in the virtual reality experience?	o yes	o no	
2.	If 1. yes, were the smells pleasant?	o yes, very	o rather yes	o no, negative
3.	If 1. yes, how intense were the smells?	o too strong	o adequate	o too weak
4.	If 1. yes, did the smells make the virtual reality experience (VR experience) more interesting?	o yes, very	o yes, a bit	o no, negative
5.	If 1. yes, were the smells at the same time as the linked events (e.g. airplane/motor smell)?	o yes, mostly	o sometimes	o seldom, never
6.	If 1. yes, were the smells adequate for the linked audiovisual events?	o yes, mostly	o sometimes	o seldom, never
7.	If 1. yes, how did you perceive the smell of the airplane/space ship?	o positively	o neutral, not	o negatively
8.	Do you think that smells could generally be an interesting component of VR experiences?	o yes, certainly	o rather yes	o no, not sure

[B] Questions on the haptic stimuli (please judge only those stimuli that were actually present)

1.	Was the physical jump exciting for you?	o yes, very	o rather yes	o no
2.	Did you notice the humidity of the clouds?	o yes		o no
3.	Was the flight through the clouds exciting for you?	o yes, very	o rather yes	o no
4.	Did you recognize the centrifugal forces in curves?	o yes		o no
5.	If yes, were the centrifugal forces realistic?	o yes, very	o rather yes	o no
6.	Did you recognize the turbulences during the landing?	o yes		o no
7.	Was the landing process exciting for you?	o yes, very	o rather yes	o no
8.	Was the flight through the building exciting for you?	o yes, very	o rather yes	o no

[C] General questions

1.	Age of the jumping subject:	o up to 18 yrs.	o 19-49 years	o 50+ years
2.	Sex/gender of the jumping subject:	o female	o male	o other
3.	Did you have VR experience before your jump?	o yes, much	o yes, a bit	o no
4.	Was our VR experience exciting for you?	o yes, very	o yes, a little	o no
5.	Did you forget the outside world during the experience?	o yes, mostly	o yes, sometimes	o no
6.	Did our VR experience cause nausea?	o yes, strongly	o yes, a little	o no
7.	How good is your sense of smell?	o very good	o good, normal	o not good

[D] To be filled out by the operating team

1.	Observed jump style:	o extraordinary	o brave	o relaxed
2.	Employed VR content:	o Skydive	o Mars	o Airrace

[E] Comments of the jumping subject: *Comments of the operating team:*

Thank you very much for your help! Horst Eidenberger, Version 2/26/2017

TU Jumpcube Questionnaire #1/2018 **English Version**

[A] Questions on the olfactory stimuli (if possible, tick the most adequate answer)

1.	Did you perceive smells in the virtual reality experience?		o yes	o no
2.	If 1. yes, how intense were the smells?	o too strong	o adequate	o too weak
3.	If 1. yes, did the smells make the virtual reality experience (VR experience) more interesting?	o yes, very	o yes, a bit	o no, negative

[B] Questions on the gustatory stimuli (if a taste stimulus was set)

1.	Did you perceive the taste during the VR experience?	o yes	o no	
2.	If 1. yes, did the taste make the virtual reality experience (VR experience) more interesting?	o yes, very	o yes, a bit	o no, negative
3.	If 1. yes, how strong was the taste?	o too strong	o adequate	o too weak
4.	If 1. yes, did the gustatory stimulus fit to other stimuli of the VR experience:	o yes, well	o rather yes	o (rather) no
5.	Do you think that taste stimuli could generally be an interesting component of VR experiences?	o yes, certainly	o rather yes	o no, not sure

[C] Questions on the haptic stimuli (please judge only those stimuli that were actually present)

1.	Was the physical jump exciting for you?	o yes, very	o rather yes	o no
2.	Did you feel the centrifugal forces in curves?	o yes		o no
3.	If yes, were the centrifugal forces realistic?	o yes, very	o rather yes	o no
4.	Did you feel vibrations caused by by-passing objects?	o yes		o no
5.	If yes, did the vibrations make the VR experience more realistic?	o yes, very	o rather yes	o no

[D] General questions

1.	Age of the jumping subject:	o up to 18 yrs.	o 19-49 years	o 50+ years
2.	Sex/gender of the jumping subject:	o female	o male	o other
3.	Did you have VR experience before your jump?	o yes, much	o yes, a bit	o no
4.	Was our VR experience exciting for you?	o yes, very	o yes, a little	o no
5.	Did you forget the outside world during the experience?	o yes, mostly	o yes, sometimes	o no
6.	Did our VR experience cause nausea?	o yes, strongly	o yes, a little	o no
7.	How good is your sense of smell?	o very good	o good, normal	o not good
8.	How good is your sense of taste?	o very good	o good, normal	o not good

[E] To be filled out by the operating team

1.	Observed jump style:	o extraordinary	o brave	o relaxed		
2.	Employed VR content:	o Skydive	o Mars	o Diving	o Airrace	o Vienna Race
3.	Employed taste stimulus:	o strong	o weak	o none		

[F] Comments of the jumping subject: *Comments of the operating team:*

Thank you very much for your help! Horst Eidenberger, Version 3/9/2018