FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# A Virtual Reality Training Tool for Upper Limb Prostheses

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Medieninformatik**

eingereicht von

**Michael Bressler**
Matrikelnummer 0425576

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Privatdoz. Mag.rer.nat. Dr.techn. Hannes Kaufmann

Wien, 29.09.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

# A Virtual Reality Training Tool for Upper Limb Prostheses

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Media Informatics**

by

**Michael Bressler**
Registration Number 0425576

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Mag.rer.nat. Dr.techn. Hannes Kaufmann

Vienna, 29.09.2013

(Signature of Author) (Signature of Advisor)

# Erklärung zur Verfassung der Arbeit

Michael Bressler
Markgraf-Rüdiger Str 3, 1150 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.

(Ort, Datum) (Unterschrift Verfasser)

# Danksagung

Zunächst möchte ich mich bei der Firma Otto Bock, und hier vor allem bei Andrei Ninu, für die Zusammenarbeit bedanken, welche dieses Projekt erst ermöglich hat. Auch gilt mein Dank vor allem meinem Betreuer Hannes Kaufmann, der mich während des ganzen Projektes mit Rat und Tat unterstützt hat.

Weiter möchte ich Cosima Prahm für die Mitarbeit bei diesem Projekt danken, so wie Christian Schönauer und Annette Mossel für die Hilfe und Unterstützung, die ich von ihnen bekommen habe.

Schließlich geht mein Dank an meine Familie und alle meine Freunde. Ihr habt alle meine Launen ertragen, und mich in jeder Hinsicht motiviert und unterstützt!

# Abstract

The technology of electromyography has become very common in being used to control hand prostheses. This technology allows the capture of controlling signals for a prosthesis by attaching electrodes to the skin, over skeletal muscles. However, before practicing with a real prosthesis, the patient has to await the healing process of the arm stump. Furthermore, the learning process can be difficult and frustrating.

In this thesis, a training environment will be presented, capable of simulating the process of grasping virtual spheres with a virtual hand by using the proper amount of grip force. The virtual reality experience, is created by using the ioTracker motion tracking system, developed at the Vienna University of Technology. This system is capable of tracking the motions of the head and arm of a protagonist with six degrees of freedom (position and orientation). The created tracking data is forwarded through the OpenTracker framework into an application created with the free version of the game engine Unity3D. In this application, the tracking data is translated into a virtual 3D environment and visualized. The picture created by the virtual camera, which is mounted to the head of the protagonist, is transmitted wirelessly to a head mounted display (HMD) that the protagonist is wearing. This allows the protagonist to move around freely inside an area of 4x4 meters.

As this work was done in collaboration with Otto Bock, the same technology was used for controlling the virtual hand, as it is embedded in the Michelangelo Hand prosthesis by Otto Bock. Using two electrodes, the electrical activity of skeletal muscles is measured through the skin and is further processed into controlling signals, which are then sent to the simulation.

As the goal of this work was both, to create an environment for exercising and to evaluate hand prostheses, the electromyographic (EMG) controlling signals can be mapped in a flexible way to certain behaviors of the prosthesis. Furthermore, several simulation modes for creating grip force can be used, which again is indicated to the protagonist by several optical grasping aides. The virtual arm can be adjusted to best match the real circumstances. Finally, several options are provided for creating and performing various evaluation and training scenarios. Based on the final application, several of these scenarios have been created and tested with probands for evaluating the capabilities of the system.

# Kurzfassung

Der Gebrauch von myoelektrischen Handprothesen ist inzwischen weit verbreitet. Mit dieser Technologie ist es möglich, Steuersignale für die Prothese mittels Elektroden zu erfassen, welche direkt auf der Haut über Muskeln platziert werden. Allerdings muss der Betroffene den Heilungsprozess der Amputation abwarten, bevor er damit anfangen kann, eine Prothese zu verwenden. Außerdem kann dieser Prozess vor allem am Anfang schwierig und frustrierend sein.

In dieser Arbeit wird eine Trainingsanwendung vorgestellt, welche es erlaubt, in einem virtuellen Raum mit einer Hand nach Kugeln zu greifen, wofür eine jeweils der Kugel enstprechende Griffkraft aufgewendet werden muss. Um die virtuelle Realität zu erschaffen, in der sich dieses Szenario abspielt, wird das Tracking-System ioTracker verwendet, welches an der technischen Unversität Wien entwickelt wurde. Mit diesem System werden die Bewegungen von Kopf und Arm des Akteurs in 6 Freiheitsgraden aufgezeichnet (Position und Orientierung) und mittels des OpenTracker Frameworks an eine weitere Anwendung übertragen, welche mit der freien Version der Game Engine Unity3D entwickelt wurde. In dieser werden diese Bewegungsdaten mittels einer dafür entwickelten Software in eine virtuelle 3D Umgebung übertragen und visualisiert. Das Bild der virtuellen Kamera, welche mit dem Kopf des Akteurs mitbewegt wird, wird drahtlos an ein am Kopf des Akteurs befestigtes Display (Head mounted display, HMD) übertragen. Dies ermöglicht es dem Akteur, sich innerhalb eines begrenzten Bereiches von 4x4 Metern frei im Raum umherzubewegen.

Da diese Arbeit in Zusammenarbeit mit Otto Bock durchgeführt wurde, konnte für die Steuerung der virtuellen Hand die gleiche Technologie verwendet werden, wie sie in der von Otto Bock entwickelten Michelangelo Handprothese eingebaut ist. Mittels zweier Elektroden wird die elektrische Aktivität von Muskeln unter der Haut gemessen und in Steuersignale umgerechnet. Diese werden anschließend über eine drahtlose Verbindung an die Simulation gesendet.

Da das Ziel dieser Arbeit sowohl in einer Trainingsumgebung als auch in einer Testumgebung für Handprothesen bestand, gibt es mehrere Möglichkeiten, die elektromyographischen (EMG) Steuersignale auf die virtuelle Hand anzuwenden. Weiters können diverse Simulationsarten für die Erzeugung von Griffkraft verwendet werden, welche wiederum dem Akteur während des Greifens durch optische Anzeigen signalisiert wird. Der virtuelle Arm kann angepasst werden, um die Simulation so gut wie möglich an die realen Gegebenheiten anzupassen. Schließlich wurden diverse Einstellungsmöglichkeiten implementiert, um das Erstellen und Durchführen von unterschiedlichen Test- und Trainingsszenarion zu ermöglichen. Im Anschluss an die Arbeit wurden Übungs-Szenarien entwickelt und mit Personen durchgeführt, Um die Fähigkeiten des

Systems zu testen, wurden im Anschluss an die Arbeit Übungs-Szenarien entwickelt und mit Versuchspersonen durchgeführt.

# Contents

CHAPTER 1

# Introduction

## 1.1 Motivation

The work presented in this thesis was initiated by the Otto Bock company, a manufacturer of prostheses. In collaboration with the Vienna University of Technology, a virtual reality application should be developed. This virtual environment should be capable of simulating the grasping action of the myoelectrical Michelangelo Hand prosthesis, developed by Otto Bock, by a virtual prosthesis in a virtual environment. Such an application could provide several benefits regarding the evaluation of prostheses by experts as well as issues of supporting the rehabilitation process of upper-limb amputees.

Myoelectric hand prostheses, such as the Michelangelo Hand have been used for more than 50 years, and since then this technology has improved steadily. Basically, such prostheses are driven by controlling signals the amputee creates by contracting and relaxing muscles in his arm stump. By using the same EMG technology, which is embedded in the Michelangelo Hand, for creating controlling signals which then are used by the virtual reality simulation it can be assured that the handling of the virtual myoelectric hand is similar to a real prosthesis.

As mentioned before, such an application could serve as an evaluation environment for hand prostheses. For example, the evaluation of a certain control mode for a myoelectric prosthesis can easily be performed without the need for programming the signal tracking hardware. For the same reason, it would also be possible to quickly adjust parameters or even try different control modes with flexibility that cannot be reached by a common prosthesis. The presence of a graphical user interface provides control over complex parameters like the hand position, which probably could not be adjusted properly without such an aide.

Furthermore, since all the action takes place inside the application, it is possible to define all parameters concerning the simulation exactly and to measure and capture every conceivable aspect of the simulated grasping action for later evaluation and analysis.

Alternatively, such an application could function as a virtual training environment for upper-limb or forearm amputees. Using computers for supporting rehabilitation issues provides all possibilities involving such a multimedia-based in- and output platform. These possibilities

range from creating a visual and acoustic output up to a highly interactive 3D gaming scenario, as known from modern video games.

The use of virtual reality should thereby allow the protagonist to reach a high level of immersion while exercising. The system presented in this work allows the protagonist to wear a head mounted display (HMD), visualizing the virtual environment. It is possible to look and move around freely (in a limited area) and in a natural way, which does not require any additional learning and allows the protagonist to concentrate on the interaction with the virtual hand.

After an amputation, the patient usually is not able to use a prosthesis immediately, since the arm stump has to heal first. Furthermore, a stem for the prosthesis has to be specifically manufactured, as it has to perfectly fit the arm stump of the amputee. During this period, it would be very useful for the amputee to have the possibility of exercising and preparing for the handling of a prosthesis. In contrast to using a prosthesis stem, for capturing the necessary controlling signals, it is usually sufficient to attach electrodes to the arm stump. Such an approach is done within minutes, and since much more flexibility is possible in regards to the healing process, this can be done much earlier than adapting a prosthesis stem.

In order to use myoelectric hand prostheses, an amputee first has to learn how to create the appropriate EMG signals. As mentioned before, this is done by respectively contracting and relaxing the particular muscle. Especially in the beginning, the use of a virtual environment can provide advantages for exercising. For example, by disabling the effect of gravity it is possible to prevent objects from falling down, which could complicate the exercises.

Depending on the accident which resulted in the amputation, as well as on the development of the healing process of the arm stump, the degrees of freedom, with wich an amputee is able to control the prosthesis afterwards, differs from person to person. The capabilities of the amputee regarding these degrees of freedom when controling a prosthesis could be evaluated and improved by such a system, without the need for a real prosthesis and therefore for a prosthesis stem as well.

Finally, the creation of serious game scenarios includes the possibility of creating highly effective exercising tasks, which are not only entertaining for the patient but also capable of maintaining the motivation to strive when exercising. This aspect certainly has a big influence on the development of the rehabilitation process. In view of modern video games, the possibilities are given to provide a varied and challenging gaming, and thus, rehabilitation experience.

## 1.2 Problem Statement

The final application, as generally presented in this thesis, is the result of two consecutive problem specifications, which will be introduced in the following subsections.

#### The Primal Problem

The first specification, given by Otto Bock, was to create a non-specific virtual reality environment, including a virtual prosthesis similar to the Michelangelo Hand as an interaction device, capable of grasping and moving objects.

A system for creating the virtual reality simulation itself already existed, therefore the decision to make use of these technologies more ore less was given. This solution consists of the ioTracker motion tracking system, developed at the Vienna University of Technology, for measuring and capturing the movements of certain targets. This captured data then is transmitted through the OpenTracker framework, and finally sent to the game engine Unity3D for creating and visualizing the virtual environment.

Thus, an application in Unity3D had to be developed, containing a virtual environment and a moveable model of a virtual hand, based on the Michelangelo Hand. As mentioned above, receiving tracking data of certain targets in Unity3D through the OpenTracker framework was already possible. These targets would be placed respectively at the protagonist's head and arm, for moving a virtual camera and the virtual arm (respectively prosthesis stem), the virtual hand is attached to.

Since the Michelangelo Hand prosthesis uses a myoelectric controlling system, the same technology was used for supplying the simulation with controlling signals for the virtual hand. In order to receive these signals from the tracking hardware device provided by Otto Bock, it was necessary to extend the OpenTracker framework by an interface to this hardware, since the interface between the OpenTracker framework and Unity3D already existed.

For mapping the received controlling signals to a certain behavior of the prosthesis, no specifications were given. In view of a highly flexible evaluation environment, an attempt was made to design a mapping system which is capable of fulfilling such requirements, as, for example, defining certain hand positions, which can be triggered by the protagonist of the simulation. As the interaction of grasping virtual objects should be as realistic as possible, the goal was to use the built-in physics engine of Unity3D to create realistic behavior of particular objects when being grasped. Designing the environment, as no specifications were given, was done with the idea of a serious game scenario.

### A More Specific Prototype

Based on this first part of the work, a more specific secondary problem statement was given. The main focus of the second specification was the simulation of grip force for evaluating feedback devices regarding this issue. Additionally, a new design for the environment, including the training objects, was given and these training object were extended by the property of only being graspable with a certain amount of grip force.

For providing this new grasping interaction, and due to problems regarding the physics engine powered grasping process, the whole interaction process was redesigned. Additionally, for supporting the protagonist in the virtual environment in creating the right amount of grip force when grasping an object, optical indicators had to be created.

Furthermore, the connection to the electromyographic tracking device for receiving controlling signals for the virtual hand had to be extended to provide bidirectional communication. Therefore these additional methods had to be implemented in the OpenTracker interface for the tracking device. Additionally, the adopted OpenTracker interface to Unity3D had to be extended by this functionality as well. In the virtual reality application itself, the functionality was implemented to send configuration commands to the tracking device.

Finally, in this second part, the concept of training scenarios was introduced in order to prepare and perform exercises. Additionally, for providing the possibility to perform various tests, primarily for evaluating the use of grip force feedback devices, several parameters regarding the new grasping interaction were designed.

## 1.3 Limitations of the Thesis

The problem of tracking human motions as performed by the ioTracker system will be introduced in the following chapter 2.2, but since this tracking system is no part of the implementation of this thesis, the technical problem of tracking itself will not be treated.

For controlling the virtual hand, the technology of electromyographie (EMG) is used for creating the relative signals. The main difficulty of this issue consists of recognizing the recorded EMG signal and creating the appropriate controlling signal, which basically is a pattern recognition issue. This task is performed by the EMG tracking device provided by Otto Bock, and will not be further treated in this thesis.

At the end of the practical work, tests were performed with four healthy people as well as with four forearm amputees. The purpose of these tests was mainly to evaluate the usability of the simulation and the practicality of the various test scenarios. However, no (meaningful) studies have yet been performed with this application.

## 1.4 Chapter Overview

This thesis is separated into seven chapters. In this chapter, the problem statements regarding the thesis were given. In the second chapter, similar research is presented, and an introduction into the main issues regarding the design and technical implementation of such an application is given. The third chapter introduces the technologies, which this thesis is based on.

In the fourth chapter, the design of the grasping interaction and of the environment is presented as well as the hardware setup for the virtual reality simulation and the scope of functionalities and settings for controlling the progress of the simulation. In the fifth chapter, the implementation of the Unity3d application as well as of the extensions for the OpenTracker framework are presented.

The sixth chapter presents the results of user-tests, performed at the end of the practical work and discusses difficulties in the implementation process as well as the usability of the environment and interaction design, also with regards to the user-tests performed. Finally, in the seventh chapter a summary of the thesis is given, and further goals and application possibilities, based on the insights and solutions gained during the process of the practical work, will be presented.

CHAPTER 2

# Related Work

## 2.1 Introduction

In this chapter, an overview of related works and approaches as found in literature will be given, on the one hand related to this work due to the technologies used, on the other hand related due to similar approaches of creating a virtual training or evaluation environment for upper limb prostheses.

The important technologies used in this work can roughly be broken down into *human motion tracking* and *controlling prostheses with electromyographical (EMG) signals*. These problems are respectively treated in the first two subchapters. The use of these technologies strongly influences the design of the resulting interaction interface, the level of immersion and finally the boundaries for creating an interactive virtual environment. This issue of interaction interfaces, as well as the continuative issue of *serious games*, will be treated in the third subchapter, both in the context of computer supported rehabilitation.

Finally, in the fourth subchapter, works with a similar approach of creating a virtual training- and evaluation environment for upper limb prostheses will be presented and compared to the requirements of this work.

## 2.2 Human Motion Tracking

One approach for creating an interaction interface for a virtual reality application is to directly use the motions of parts of a human body and translate them into the virtual environment.

For tracking human motions, a huge range of technologies has been developed, using a wide range of different technical solutions. A general classification of these tracking technologies is given in [31] (fig. 2.1). As illustrated in this figure, another subdivision is done into *visual* and *non-visual* technologies. Searching in literature for recent works with the similar goal of tracking human motions in the domain of computer-assisted rehabilitation predominantly returned approaches using visual tracking technologies, again in a large variety of implementations.

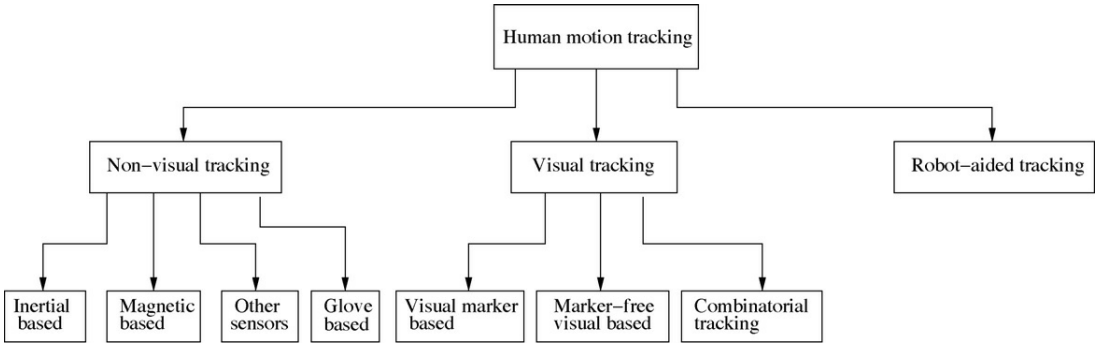



**Figure 2.1:** A general classification of human motion tracking technologies given by [31].

| System | Accuracy | Costs | Drawbacks |
|---|---|---|---|
| Inertial | High | Low | Drifts |
| Magnetic | Medium | Low | Ferromagnetic materials |
| Ultrasound | Medium | Low | Occlusion |
| Glove | High | Medium | Partial posture |
| Marker | High | Medium | Occlusion |
| Marker-Free | High | Low | Occlusion |
| Combinatorial | High | High | Multidisciplinary |
| Robot | High | High | Limited motion |

**Table 2.1:** Performance comparison of different motion tracking systems according to fig. 2.1 ( [31], in condensed form).

Due to the rising capability of real time video recording and processing, especially in the domain of smart phones, the costs for this type of technology is getting lower and the field of possible applications is getting bigger. Visual tracking therefore seems to be the most recent State-Of-The-Art technology and since this work also uses a marker-based visual tracking system, the focus in the following will be on visual tracking solutions.

A proper low-cost visual solution without using markers is the Microsoft Kinect sensor, which provides a real-time recognition and tracking of up to two protagonists. In [16] such a skeletal tracking solution is presented. This technology is not expensive and even sufficiently accurate for moving a virtual prosthesis around in a simple training environment, but the interaction method is more suitable for a visual output on a screen or television than on a head mounted display as used in this work, since the head motions are not translated to the virtual environment. Nevertheless, for rehabilitation issues as well as for virtually evaluating prostheses such an approach might be sufficient.

Another interesting visual approach for tracking human motions without using markers is presented in [24]. Cameras are placed on certain points of the protagonist's body and their captured material is later used for calculating a three-dimensional skeleton model (see fig. 2.2). This technology was developed for motion capturing with the goal of character animation and is

not suitable for this work, at least because of the expensive nature of implementing wireless real-time measurement for all the cameras. However, by comparing this and the previous technology, another interesting criteria for visual tracking systems can be introduced.

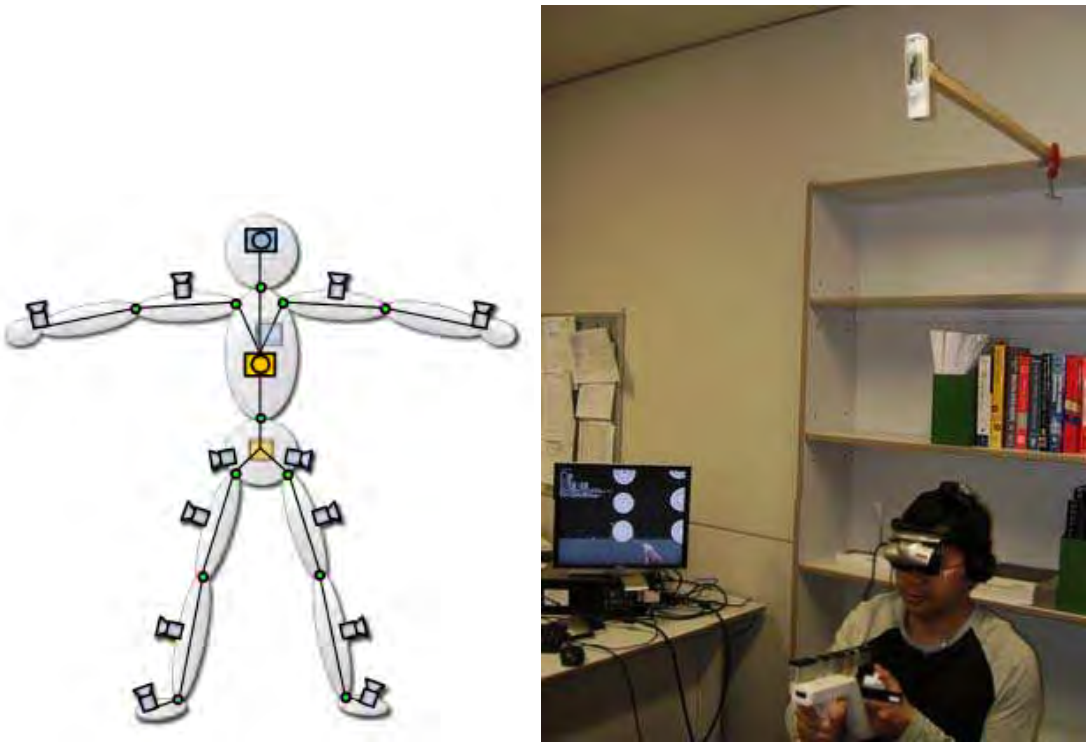


**Figure 2.2:** Left: The protagonist is wearing body-mounted cameras for motion capturing [24]. Right: A overhead-mounted Wii Remote control is tracking the positions of the protagonist's head and of the hand-held second Wii Remote control. The joystick of this second control is used for spatial navigation [5].

While the second solution requires the protagonist to be equipped with cameras, which after being placed need to be calibrated, the Kinect sensor needs no additional equipment and the protagonist can start interacting without any preparations, which - in relation to rehabilitation applications - can be displeasing for the protagonist. Furthermore, the personal moving space is not limited (even not mentally) when not wearing any equipment.

A setup better fitted to the requirements of this work is presented in [5]. This approach uses two Wii Remote controls. One is held by the user and moved around as virtual input device (additionally, the joystick of the Wii Remote is used for spatial navigation in the virtual environment), while the other control is fixed above the head of the protagonist and used for tracking the positions of the head and the handheld-control. For tracking, the infrared sensor of the Wii Remote is used, and clusters of infrared lights are mounted on the protagonist's head as well as on the hand-held Wii Remote (see fig. 2.2).

Since this technology can be classified as marker-based visual tracking, and as the solution used in this work falls into the same category, a closer overview of works using this technology will be given in the following subchapter.

### 2.2.1 Marker-Based Visual Tracking

Since the goal of this work was to create a realistic simulation of controlling a prosthesis, the decision was made to achieve this goal by using the technology of electromyography (see chapter 2.3 for an introduction). For controlling a virtual hand by using a healthy hand, an approach is presented to track the movements of the hand by using a colored glove [30] (see fig. 2.3). This solution is interesting insofar as the use of colored labels could also be extended to the whole body and be used for tracking the head's position as well. This is important for the virtual HMD experience.

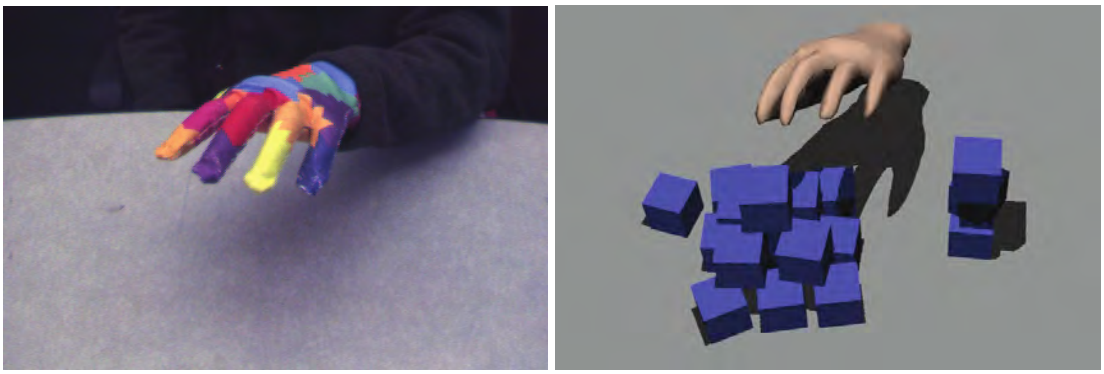


**Figure 2.3:** A colored glove is tracked by a camera for controlling a virtual hand [30].

Following this last idea of tracking the whole body, a similar approach is presented in [6]. This system again uses infrared LEDs, but unlike to the Wii Remote setup, the LEDs are mounted to the camera instead of the protagonist. The protagonist on the other hand wears five infrared-reflective markers (ankles, wrists and belly). Since a marker would be occluded easily when only one camera is monitoring the scenery, four cameras are placed in a way that they monitor a certain area in which the protagonist can move around. This technology is close to the ioTracker system used in this work.

Using the tracking system as presented in [6] allows the tracking of position of certain points of interest, such as the ankles, wrists and the belly of a protagonist. This allows the creation of a rough 3D skeleton model, with the level of detail of this model can easily be increased by adding additional markers (e.g. elbows, knees). For each marker, the position in space can be calculated for all three axes, which is consistent with three degrees of freedom (3-DOF). But for creating a virtual HMD experience, it is necessary to track the heads position as well as its orientation, which extends the required degrees of freedom to six (6-DOF).

This can be achieved by tracking several points for each marker. While one point per marker allows 3-DOF, with two points 5-DOF and with at least three points 6-DOF are possible to track [8] (see fig. 2.4). Similar to this approach, the ioTracker system used in this work uses

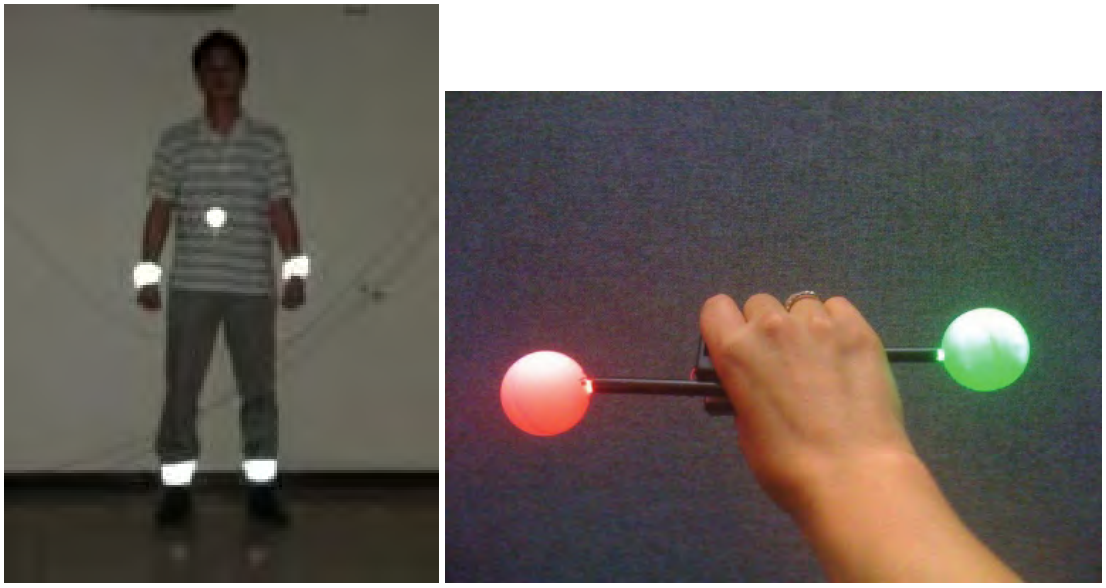


**Figure 2.4:** Left: POSTRACK protagonist wearing five retro-reflective markers [6]. Right: Tracking one, two ore three points provides information of three, five or six degrees of freedom [8].

multiple-point clouds per marker for achieving 6-DOF tracking per marker. An introduction to the ioTracker system can be found in chapter 3.2.

## 2.3 Electromyography (EMG)

As mentioned in the first chapter, one goal of this work was to create a prosthesis control, which is as realistic as possible. This was done by using hardware developed by Otto Bock and used in their prostheses. This hardware makes use of the electromyography (EMG) technology, which will be introduced in the following.

The history of myoelectric hand prostheses started more than 50 years ago, and the underlying surface EMG technology basically measures the electrical activity in skeletal muscles. This requires no surgery for the protagonist but only electrodes attached to the skin over the respective muscle (see fig. 2.5). Furthermore a relatively small muscle activity is sufficient for creating controlling signals. Such electrodes can also be mounted inside the stem of a prosthesis, which guarantees correct positioning and allows the device to be easily taken on and off [13] [23].

The measured signal created by such an electrode is a high-frequency noise. By using a threshold value, this noise can be translated into a simple on/off signal as illustrated in fig. 2.5. For being able to retrieve more information out of this raw EMG noise signal, an early approach was to use pattern recognition. The first project using pattern recognition was started in the early seventies and used perceptron classifiers for controlling a hand prosthesis [23].

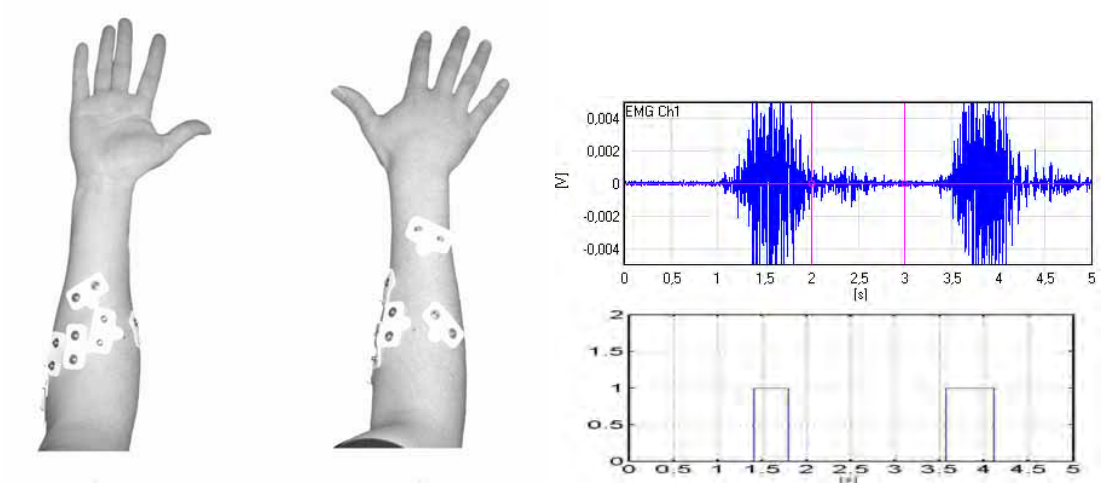



**Figure 2.5:** Left: EMG electrodes attached to the lower arm [23]. Right: A recorded EMG signal (top) and the corresponding control command generated by a threshold (bottom) [13].

A more recent approach of pattern recognition is presented in [22]. In this project 16 electrodes are attached to the skin of the forearm[1]. This setup aims to reduce the noise in the recorded signal to a minimum. For feature extraction, the signal is separated into its underlying basic frequencies. Classification is done by collecting these frequencies together into a feature vector and compare them to a database with already classified feature vectors. Another algorithm for classifying EMG signals is presented in [13] and is based on a neural network in combination with an auto-regressive based algorithm for feature extraction, in order to keep the needed processing power low.

More examples for the use of EMG technology in rehabilitation and prosthetics are given in [25], where a EMG controlled voice prosthesis is presented which uses pattern recognition for classifying the recorded signals as single words. For post-stroke hand rehabilitation, a EMG-driven hand robot is presented in [14]. This robot is steered by two EMG electrodes, which are attached to the forearm. The electrode for opening the hand is located on the extensor digitorum (ED) muscle, which is usually used for extending parts of the hand. For closing the hand, the second electrode is located on the abductor pollicis brevis (APB) muscle, which is used, for example, when grasping.

Such a controlling system consisting of two electrodes is similar to the one used in this work. The use of only two electrodes limits the degrees of freedom for controlling the prosthesis. However, when measuring more than two signals, things soon start to get complicated. If the electrodes are located too close, it is not possible to detect the proper muscle activity with the respective electrode. As experienced during this work, this can even be a problem when using only two electrodes. Another issue is the ease of learning for the user. Concerning amputees, among other criteria it depends on the shape of the prosthesis stem, as resulting from the needed surgeries, and also on the age of person involved, how many degrees of freedom can be handled

[1] Additionally to the 16 surface EMG electrodes, six needle electrodes are attached

by the user. Finally, opening and closing the hand prosthesis seems to be sufficient for fulfilling most of the essential daily actions and restore independence and quality of life for the amputee, which is another important criterion for finding the right balance between easy usability and sufficient flexibility.

### 2.3.1 Control of a Virtual Hand

In contrast to controlling the virtual hand with visual tracking technologies, such as a colored glove [30], in the work presented in [10], not only the movement of the virtual hand itself are tracked by EMG signals. Supported by accelerometers, the EMG signals are classified into several kinds of hand motions and arm positions, and therefore also allow to approximately track the position of the arm.

This is achieved by using a wristband of electrodes for generating better results in pattern recognition, and as mentioned before two accelerometers are placed in the middle of the upper arm and the lower arm. For classifying the limb positions and hand movements, a database was created for later being able to recognize eight types of hand motion and 5 arm positions (see fig. 2.6).

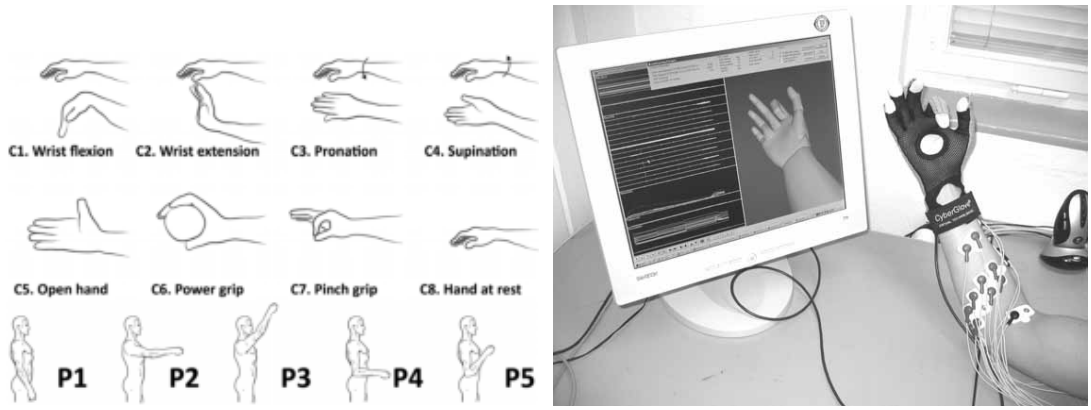



**Figure 2.6:** Left: The EMG tracking system presented in [10] is capable of classifying eight types of hand motion (C1-8) and five different arm and elbow positions (P1-5). Right: A data glove is used for measuring the hand position and creating relations among hand positions and the measured EMG signals for later classification in [23].

In [23] a similar approach is described. In this work the healthy hand of the protagonist gets equipped with a data glove, capable of measuring the movements of the fingers. Additionally, up to 26 electrodes where attached to the forearm for measuring the appropriate EMG signals for the respective movement of the hand or the fingers (see fig. 2.6). The use of a data glove makes it easy to define features for several positions in a highly accurate way. This approach is interesting insofar, as it is a real-time approach and suitable for being used for controlling prostheses. According to the paper, all tested subjects were able to manage at least eight of the nine possible movements 100% correctly after two hours. This is an impressive result and shows a promising perspective for future prostheses.

## 2.4 Computer-Assisted Rehabilitation

As the domain of computer-assisted rehabilitation is huge, the overview of related papers given in this section will be limited to upper limb rehabilitation applications. The most interesting aspect of such a rehabilitation system probably is the design of the interaction interface, as this determines the kind of exercises which can be performed. Furthermore, limiting the works makes it possible to compare the particular interfaces presented, as the requirements for creating upper limb rehabilitation environments basically include the control of a virtual hand and/or arm[1]. This issue will be treated in the first subsection.

The effectiveness of a rehabilitation program obviously can be increased by raising the level of active participation of the patient. Especially for elderly patients the motivation to eagerly perform an exercise can be improved by providing interesting and entertaining training exercises [9]. Based on the interface, which in large part specifies the level of immersion that can be accomplished by the user, this is another interesting aspect of computer-assisted rehabilitation, as the use of computers makes it easy to create quite complex multimedia-based interactive scenarios. An overview of *serious games* regarding upper limb rehabilitation will be given in the second subsection.

### 2.4.1 Interaction Interfaces

The use of computers for rehabilitation does not only give access to a multimedia-based platform, it also opens the door for the use of telecommunication. This enhances the level of convenience for patients with reduced mobility, while it also enables autonomous exercising and allows a more efficient coaching of patients by the clinic.

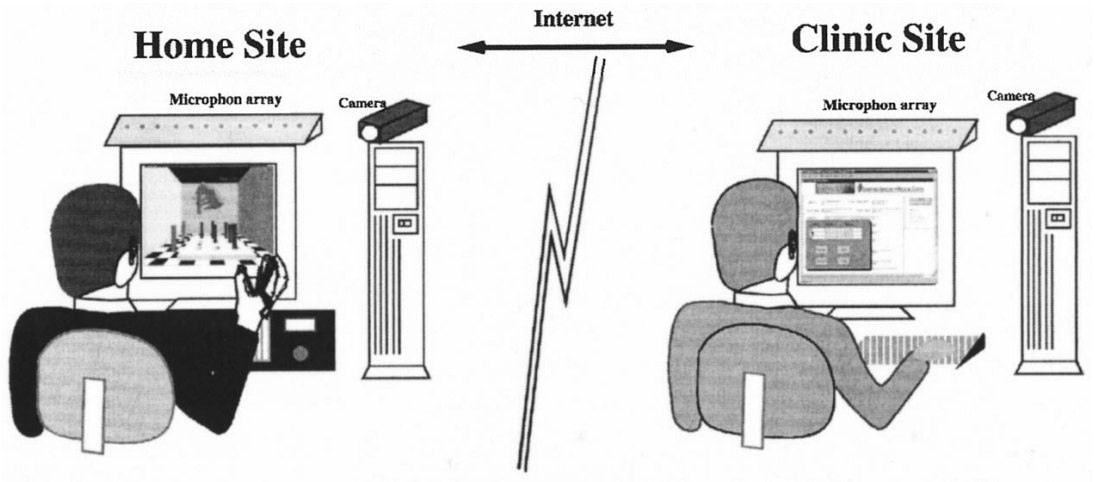



**Figure 2.7:** A system for telerehabilitation as presented in [2].

[1] The most interfaces which are considered for being used with a healthy hand, can also been thought of as closing and opening of the virtual hand is being controlled by the use of EMG electrodes attached to an arm stump.

This approach was presented in 2000, therefore the technology used for the interface is not state-of-the art. Nevertheless, it is interesting to have a closer look at it. The user wears a glove for tracking the motion of the fingers. A receptor for a magnetic tracking system is attached to the wrist of the user in order to track the position of the hand. For voice commands, a microphone array, connected to a voice recognition software running in the background, is placed on top of the screen. Finally, a camera is installed with patient as well as in the clinic for video telephony (see fig. 2.7) [2]. This interface, if slightly modified, basically would also fit the needs of this work, and the concept of telerehabilitation is promising.

Another kind of interaction interface, which is rather present in works for upper limb rehabilitation, is the use of a robot arm, as presented in the rehabilitation system *PLEMO* [17] (see fig. 2.8). Basically, the benefits of such a system in contrast to a visual tracking system are the lower costs of calculation power needed for the process of tracking. However, for controlling the position of a virtual arm and hand, such a system might limit the free moving space of the patient too much, resulting in the danger that the exercises performed are too grinding and onesided. Furthermore, because the accuracy of visual tracking systems is increasing, the benefits of using robot arms as the less complex technical solution, is disappearing.


**Figure 2.8:** Left: PLEMO Rehabilitation System for Upper Limbs. The patient has to move the grip over the working table, according to the task, given on the screen [17]. Right: The user is wearing a HMD and a glove for interacting with a virtual environment [20].

An interaction interface close to the one used in this work is presented in [20]. The patient wears a HMD display and a glove. The positions of the head and the glove are tracked and enable the patient to look around in a 3D environment and use the hand as interaction device (see fig. 2.8). Additionally, the glove measures the position of the fingers for controlling the virtual hand. This interface setup provides a high level of immersion due to the HMD and the

intuitive control of the virtual arm and hand, and is therefore a good foundation for creating exciting and entertaining rehabilitation exercises.

### 2.4.2 Serious Games

As mentioned in the beginning of this subchapter, the use of games for rehabilitation issues has clearly proven its ability to increase the patient's motivation during exercising. While creating stimulating games for elderly people is a greater challenge, for children such a virtual interactive gaming environment can be satisfying over a longer period of time [11] [9].

An important factor for creating a satisfying gaming experience is the use of scoring mechanisms. These not only allow to monitor the progress of the rehabilitation process, but also give the user the possibility to find a challenge in breaking a new high-score. And finally, a scoring system makes it easy to create a meaningful game, which also is an important factor for motivation [3]. Especially at the beginning of the rehabilitation process, the experience of failing can be quite frequent for the patient. To avoid this, games can be designed in a way that they adapt the level of difficulty to the skills of the player [4]. This is a very common approach in the domain of entertainment games, since it is usually necessary to introduce a player to the gaming process at the beginning. By, for example, separating the process of playing into several stages and respective levels, a straight and comprehensive path during the game and the rehabilitation process can be created for the patient. In figure 2.9, two examples for upper limb rehabilitation games are illustrated.


**Figure 2.9:** Two examples for *serious* games used for upper limb rehabilitation. Left: Whack a mouse game, Right: Catching oranges game, both presented in [3].

The exercises themselves, seen from the perspective of rehabilitation, can also be divided into several different gaming tasks for keeping the patient's motivation up. Therefore, exercising a grasping process could start with catching a mouse on a table, while the next level in difficulty could be grabbing oranges from a tree, which requires the patient to stand straight and move the arm as well. A third level of difficulty could be the catching of some sort of moving objects. As long as playing the game creates satisfaction for the patient, it will support the rehabilitation

process. Scenarios as illustrated in figure 2.9 are neither costly to the hardware needed for visualization, nor does their creation require much time. By using technologies for the domain of entertainment games, it is easy nowadays to create visually appealing games which are able to provide a huge variety of scenarios to explore for the player.

## 2.5 Virtual training environments for upper limb prostheses

In this section, papers with a similar goal of creating a virtual training and also evaluation environment for upper limb prostheses are presented. In all of the following projects, the virtual hand is controlled by using EMG signals, the main difference among the particular works lies in the level of immersion, determinated by the design of the (visual) feedback technology chosen, and the tracking technology used for tracking the head and arm positions of the protagonist.

The first work presents a simple training environment, consisting of EMG electrodes attached to the upper forearm of the protagonist, which sits in front of a monitor and watching a virtual hand, which is controlled by the generated EMG signals [19] (see fig. 2.10). As no tracking of the hand or arm position is done, the only possible exercising scenario for the patient is to watch a virtual illustration of the hand (or something else) opening and closing. In view of keeping the user motivated, this approach might be sufficient for a clinical evaluation software, but not for long-term rehabilitation exercises.

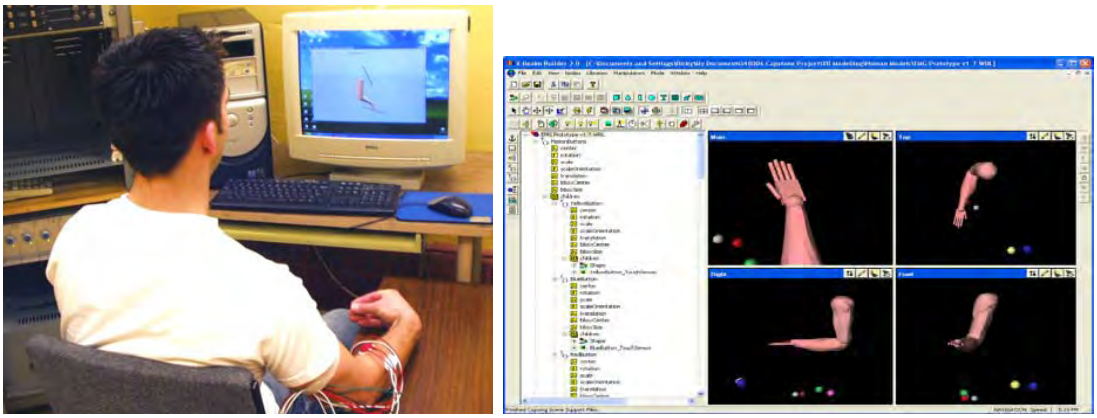
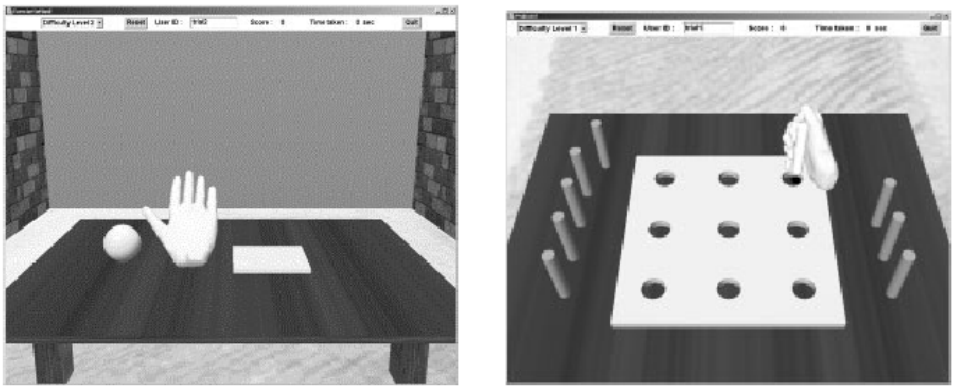


**Figure 2.10:** Left: A very simple setup for just controlling the closing and opening motion of a virtual hand as presented in [19]. Right: This setup is more complex as it allows to control the complete arm starting from the shoulder. The task in this environment is to grasp the colored balls [1].

A similar approach is presented in [1], which provides a higher level of freedom by giving the patient control of the whole arm. This is not done by (visual) tracking, but also via EMG control signals. To achieve this, EMG electrodes are attached to the particular muscles of the patient. The task given in this environment is to grasp and release balls by creating the appropriate EMG signals (see fig. 2.10), which is rather similar to the approach in this work. (Furthermore, it is notable that the design of the environment provided in [1] is very similar to

the final design of the environment in this work (a black background and colored balls as objects to be grasped)). Another approach for controlling the arm's motion is presented in [27], where electro-goniometers are attached to the patient's shoulder and elbow to measure the angle and determine the position of the virtual hand. The hand itself is controlled by EMG electrodes attached to the forearm.



(a) (b)

**Figure 2.11:** The patient is wearing a HMD and controlling a virtual hand by creating EMG signals. The exercies task is to pick up cylinders from a table and to up them into the holes [18].

A system providing even more freedom in moving the virtual hand is presented in [18]. The task given is to grasp cylinders with a virtual hand and put them into holes (see fig. 2.11). For recognizing the several arm positions, the system first has to be trained by an avatar performing certain movements, which then have to be repeated by the patient. For creating the visual output, the patient wears a HMD, and the position of the head is tracked. The virtual environment is built in VRML (Virtual Reality Modeling Language) and visualized by using a web-browser.

The interface presented in [12] is very similar. The patient wears a HMD for visual output, the tracking of the arm is achieved by gyro-based sensors attached to the shoulder, the elbow and the wrist. For tracking the head position and the orientation, the same type of sensor is used. For controlling the virtual hand, EMG signals are created by electrodes placed in the wristband. The system is thought to be run on two computers, one for rendering the visual output for the HMD, the other one is operated as an usual desktop pc and functioning as a second eye into the virtual environment as well as a control platform for tasks performed. The task in this work was to grasp a random virtual cube, move it to a virtual box on a desk and release it [12] (see figure 2.12).

Compared to this work, all approaches presented in this chapter provide less freedom of moving for the patient, regarding the virtual environment as well as the moving of the arm and thus also the virtual prosthesis. Depending on the kind of rehabilitation exercise, a limited level might be sufficient, but in light of the creation of exciting interaction scenarios for serious games the approach used in this work might be better. For controlling the virtual hand with respect to the prosthesis itself, several approaches were presented which provided greater access to the

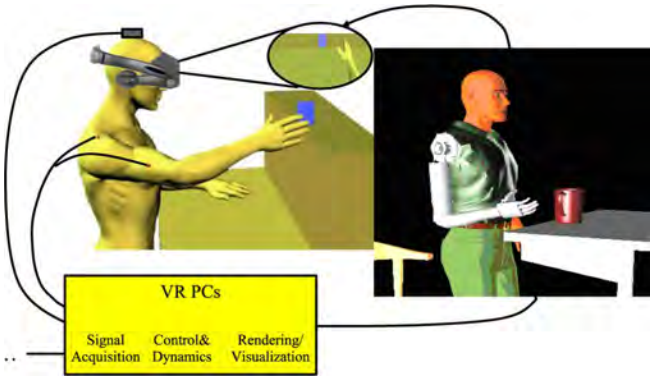



**Figure 2.12:** The user is wearing a HMD and controlling a virtual hand and arm by EMG signals respective by gyro-based sensors. The task in this application is to pick up virtual objects and put them in a virtual box [12].

virtual hand than in this particular work by giving access to the particular fingers. This is mainly due to the collaboration with Otto Bock and the initial goal of creating a simulation based on their prostheses. However, the system presented in this work easily allows to change this detail if necessary.

CHAPTER 3

# Theoretical Foundations

## 3.1 Introduction

As mentioned in the previous chapters, for tracking the position and the orientation of the protagonist's head and arm this work uses the visual, marker-based tracking system *ioTracker*. An introduction to ioTracker will be given in the following section. The second section will introduce the EMG tracking device, developed by Otto Bock. This device is used for receiving controlling signals for the virtual hand. For creating the virtual environment and the interactive interface, the game engine *Unity3D* was used. The basic concepts of this engine will be introduced in the third subsection. Finally in the fourth subsection the OpenTracker framework is introduced, which is used for providing a real-time data exchange among the particular tracking components and the applications providing the virtual environment.

## 3.2 ioTracker

The ioTracker [15] system was developed by members of the Virtual Reality Research Group at the Vienna University of Technology and has the advantage of being a highly accurate and efficient low-cost tracking system. Referring to the classification given in [31], ioTracker is a visual marker-based tracking system (see chapter 2.2). Each target can be tracked with 6-DOF (orientation and position) and it is possible to use up to 12 tracking targets at the same time. For tracking the protagonist's head and arm, this is more than sufficient.

For observing a certain region, ioTracker can be used with four, six our eight installed cameras, which shape the observation space (see fig. 3.1). Before using the system, the cameras have to be calibrated for exact estimation of the position and orientation for each of the cameras. After calibration, even the smallest change in position or orientation to the cameras would lead to an inaccurate tracking process.

Each of these cameras is equipped with an infrared spotlight and uses an infrared-filter attached to the lense. The reduction of the captured image to infrared light increases the

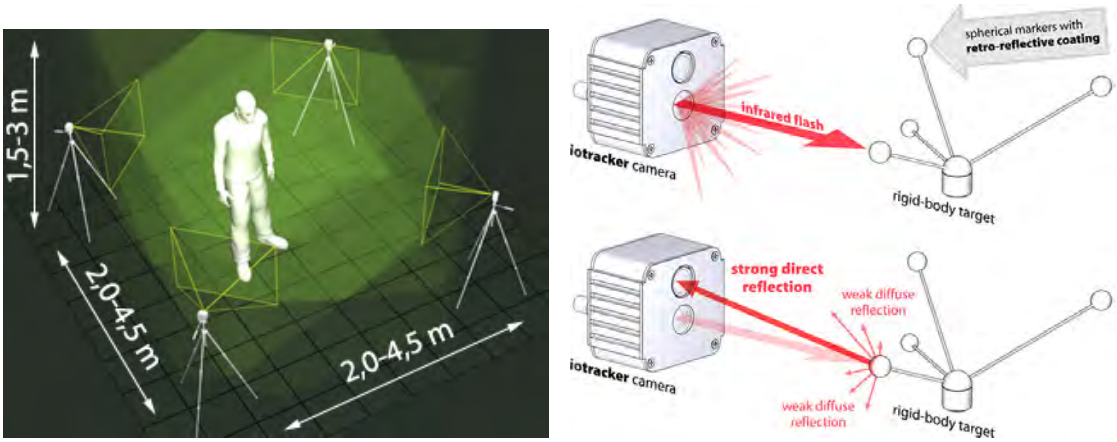



**Figure 3.1:** Left: Tracking space shaped by 4 ioTracker cameras. Right: Functioning of an ioTracker camera.

effectiveness of the tracking process, since the information which has to be processed by the tracking system is significantly reduced. The tracking targets used by the ioTracker system consist of several small spheres, which are covered with a retro-reflective surface[1]. These spheres are aligned in a way that they form a rotation-invariant point cloud, which allows to determine the exact position and orientation in space. In the ideal case, due to the infrared filter attached to the cameras, each camera retrieves a black image with only the spheres of the markers used as white objects. Such a lightweight tracking target can easily be attached to almost any object and thereby enables this object to function as virtual interaction device. For example a pen or a clipboard can be translated into a virtual environment by translating their position and orientation to virtual objects and thereby enabling the protagonist to interact with them in the virtual world. Similarly, a tracking target attached to a head-mounted display (HMD) can turn this display into a virtual camera, generating an image of the virtual environment (see fig. 3.2).

As one can imagine, the process of tracking has to be highly accurate. If the virtual hand does not follow the movements of the real hand with sufficient accuracy, it would be difficult to reach for an object. Furthermore, if the position of the virtual camera did not follow the motion of the head, this could lead to so-called simulator- or cyber-sickness.

For tracking the position and orientation of a tracking target, technically images from at least two cameras are required, while the minimal setup for an ioTracker system uses four cameras. The more cameras are used, more accurate and stable results can be calculated by ioTracker. Another important reason for using eight cameras, is to prevent masking of the tracking targets. This can easily occur - for example if the protagonist holds the arm (stump), attached with a tracking target, in front of his body. This would mask the target for all cameras, which are located behind the protagonist. Since masking a tracking target would lead to an absence of

[1] A retro-reflector is able to reflect the impacting light back to the emitter with a minimum of scattering.

**Figure 3.2:** Two examples of ioTracker tracking targets, attached to a HMD (left) and a pen (right).

tracking data, which is needed for real-time synchronizing the virtual prosthesis and even more important the virtual camera[1], this situation should be avoided.

Even though the cameras of the Opentracker system only detect the reflected infrared light, tracking errors could occur if any surfaces other than the spheres of the tracking targets reflect infrared light to only one camera. Therefore, for achieving the best tracking results, all reflecting surfaces have to be covered or hidden from the cameras. As daylight also contains infrared radiation, it should be avoided by darken the room the ioTracker system is set up in, which has the additional benefit of creating a better contrast for the protagonist wearing the HMD.

For distributing the generated tracking data over a network or to other applications, ioTracker provides an interface to the OpenTracker framework, which will be introduced in chapter 3.5.

## 3.3 Otto Bock EMG Tracker

An introduction to electromyography (EMG) is given in chapter 2.3. In short, electrodes are attached to the skin of the protagonist to measure the electrical activity of the underlying skeletal muscles. This makes it easy to (dis-)connect such a device without the need for a surgical procedure. The disadvantage is the inaccuracy and unsteadiness of the measured signal and requires an efficient and complex correction of the signal. By using technologies like pattern recognition, the signal recorded by the electrodes can be translated into a controlling signal. The stronger a muscle is contracted, the higher the resulting controlling signal. This process is a challenge of its own and has been omitted in this work by using the EMG tracker technology of Otto Bock. This, furthermore, has the advantage that the reactions of the virtual prosthesis are quite similar to, at least, a prosthesis made by Otto Bock.

The EMG tracking device used in this work is embedded in a wristband, together with two EMG electrodes for tracking the signals for opening and closing the hand. This setting is similar

[1] Masking a target can cause slight jittering of the camera or arm up to rapid jumps, which can cause cybersickness.

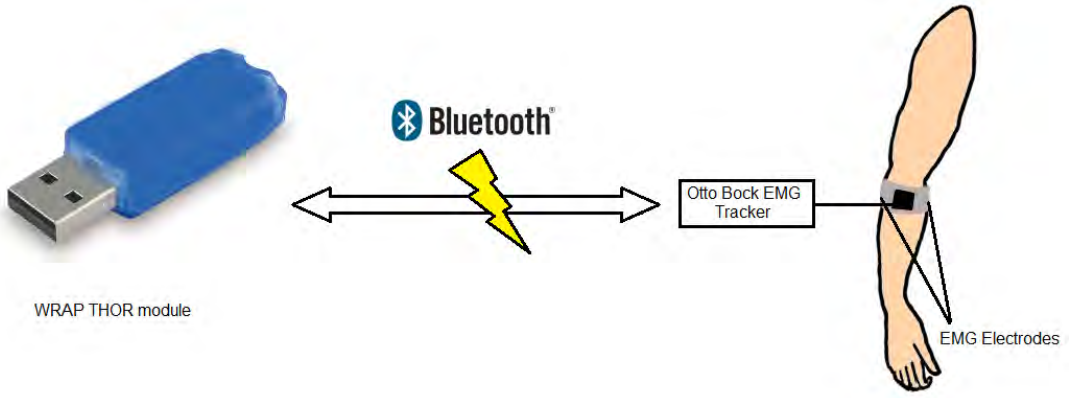



**Figure 3.3:** An illustration of the OttoBock EMG tracking device, embedded in a wristband. Communication to the computer is established via the bluegiga WRAP THOR module, which is capable of connecting to another Bluetooth device (the EMG tracker), and accessable from the computer via a virtual serial port. For capturing the EMG signals, two electrodes are embedded in the wristband as well and connected to the tracking device.

to the controlling electrodes in the stem of a myoelectric prosthesis. For communicating with the computer, a WRAP THOR module is used, which provides a virtual serial port at the computer for sending and receiving bytes. Furthermore, this interface implements the Bluetooth protocol and is able to connect to any Bluetooth device. This allows a wireless communication between the computer controlling the simulation and the protagonist wearing the EMG tracking device.

For sending and retrieving data, the OpenTracker framework was extended by an interface(module) capable of establishing a connection through the serial port to the WRAP THOR module, and furthermore to the EMG tracking device. The OpenTracker framework will be introduced in chapter 3.5, while a detailed description on the implementation of the OpenTracker interface for the EMG tracking device is given in chapter 5.4.

## 3.4 Unity3D

The Game engine Unity3D[1] is a complete developing environment, providing possibilities for creating 3D scenes with objects and lights by clicking and dragging, much the same as 3D modeling software, but with the exception that 3D objects cannot be created or modified. This makes it very easy to set up or rearrange a 3D scenery. But in contrast to modeling software, each object can contain several scripts which are capable of modifying all of the object's scripts as well as other objects respectively their components. For example, such a script could be used for opening a door when entering a certain area. Thereby it is possible to make the scene interactive.

[1] `http://www.unity3d.com` [28]

This is done by combining the objects in the scenery with scripts which literally bring them to life. The system of managing these objects and scripts is introduced in the first three subsections.

Unity3D comes with the built-in physics engine PhysX from NVIDIA. This actually influenced the process of solution finding of this work, as a first approach of creating the grasping interaction by making use of the physics engine was not satisfying and finally led to a simplified interaction process. The physics engine will be introduced in subsection 3.4.5.

Concerning multiplayer gaming, Unity3D uses the open source multiplayer game network engine RakNet. This engine makes it possible to use the objects as mentioned above for directly communicating over a network with other objects from another application. The technical aspect of using the network engine of Unity3D will be explained in detail in the last subsection.

For performance issues, each application built by Unity3D automatically provides options for resolution as well as for rendering quality regarding textures, lighting and shaders. This easily allows to adapt the performance required of the simulation to the capacities of the computer.

Especially for prototyping, but maybe also for creating the final software, using Unity3D safes a lot of time since a lot of functions needed to create a VR simulation are already provided and experimenting as well as debugging is very easy due to the graphical interface of Unity3D and the possibility of real time editing. However, as Unity3D is a game engine, some special requirements like an extended window and widget system might not be fulfilled. Tasks which require real time or at least an accurate time control might not be possible to implement due to the fundamental Unity3D processes in the background which cannot be influenced. Therefore, it might be useful to prototype applications in Unity3D, but use less dynamic and powerful development environments (if at all) for creating the final product. With an appropriate approach, a lot of the written code could even be reused.

### 3.4.1 GameObjects

A GameObject is the base class for each entity used in a Unity3D scene [29]. It contains a name which does not have to be unique and a Transform component, which defines the objects position, orientation and scale in the scenes coordinate system. Furthermore, GameObjects can be tagged and applied to various layers which themselves can be defined in Unity3D. In this work this is used, for example for providing different rendering contexts inside one Unity3D scene. Another important functionality of GameObjects is their capability of being hierarchically composed. Each GameObject can contain other GameObjects as children. If a GameObject is a child of another GameObject, its Transform component's position, rotation and scale are relative to the position, rotation and scale of the parent's Transform component. Only if a GameObject has no parent, its Transform values are global[1].

The real power of GameObjects is due to the fact that they contain components. As mentioned above, each GameObject contains a Transform component which manages the GameObjects position, rotation and scale. This component is a fundamental requirement for

[1]Meaning that they are relative to the scene's origin and rotation, both zero vectors and the original scale of one in each dimension.

a GameObject and cannot be deleted. But in addition to the Transform component, any number of components can be added to this GameObject.

### 3.4.2 Components

A component is basically a script and can be applied to a GameObject. There are several types of premade components for any purpose, like *Mesh Filters* for holding geometry data, *Mesh Renderers* for rendering the geometric data provided by a Mesh Filter with a certain texture, *Cameras* for providing view ports the scenery can be rendered through, *Lights*, *Particle Systems*, *Physics components*, *Audio components*, *Networking controls* and many more. For each component, a setting interface is provided by the Unity3d editor for adjusting the component as needed.

One benefit of components is that they can also be created from scratch by the developer. Each component (script) is derived from the class *MonoBehaviour* and provides several overrideable functions which are accessed by the main loop of the Unity3D application [29]. The most important ones to be mentioned are the *Start* function, which is executed after loading the application and can be used for an initializing process, and the *Update* function, which is executed every frame.

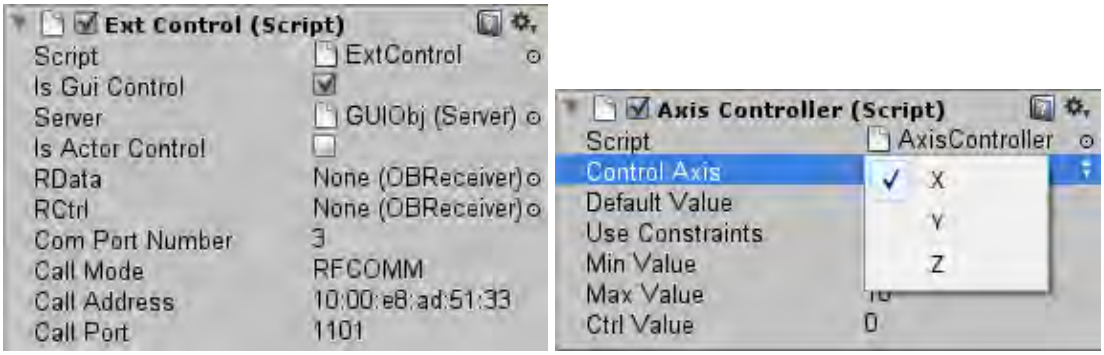



**Figure 3.4:** Examples for setting interfaces of self made components.

Another very useful feature of the self made components is their capability of providing a setting interface for the Unity3D editor like the premade components do (see figure 3.4). In game developing this allows the level designers and artists to work with components at a "higher" level with no need to be confronted with the written code. Based on this idea, and assuming that the "experts" using the simulation are skilled in using Unity3D, this would allow a high degree of flexibility in creating testing environments and other adjustments to the scenery. Without using the Unity3D editor, this would require a complex user interface inside the application. The settings interface can be easily defined by simply defining public variables for the derived MonoBehaviour class. Each public variable is displayed in the settings interface according to its data type. Therefore, text has to be typed, while number values can be typed, but also defined continuously by clicking and sliding. Boolean expressions are displayed as checkboxes, pointers to other classes (components) provide a window for selecting the appropriate components or can

be defined by directly dragging another component onto this parameter and even enumeration data types are possible to use as parameters, providing a drop-down menu with the respective items.

Finally another feature of components is the possibility to define requirements of other components for being used. For example, if a component has to communicate via network, a NetworkView component is required (see 3.4.6) and ideally should be applied to the same GameObject as the other component. Such a requirement for the NetworkView component can be defined by adding the line *[RequireComponent (typeof (NetworkView))]* directly in top of the class declaration. This also works with self-made components and has the effect that if such a component is applied to a GameObject in the Unity3D editor, the other components required are automatically created and applied to the GameObject as well. This makes it much easier - especially for non-expert stuff like artists and level designers - to stay on top of things.

### 3.4.3 Prefabs

Prefabs are used to save and load hierarchies of GameObjects as they are created in the Unity3D editor. Each of the GameObjects in the hierarchy can contain several components with their parameters specified. Even saving a whole level as a prefab would be possible. However, in this work no levels are required, and prefabs especially are used for multiple used GameObject hierarchies like the virtual arm containing the prosthesis with all its individual parts. This structure all in all is used four times[1], another example are the Training Objects, which are used for being grasped by the protagonist. When changing a prefab or a component of a prefab, each occurrence of this prefab in any scenery saved is also affected. If the user changes parameters or the structure of a prefab, which is used in the scenery loaded in the editor, the hierarchical structure gets detached from its binding to the prefab and is not affected by changes of the prefab anymore.

Since prefabs can help to save time in the process of level creation, this would already be a good reason to use them. However, another even more important reason is the fact, that prefabs are intended for being created and destroyed at run-time. Each prefab can be used for creating instances of itself during runtime. This allows to create dynamically huge amounts of objects like trees or - as it is required for this work - Training Objects.

### 3.4.4 Assets

For including any form of content into a Unity3D scenery, which cannot be loaded directly by components, assets are used. An asset can contain scripts, shaders and textures, but also prefabs, which again contain GameObjects with components. Generally, an asset can be considered a container for embedding resources, which are saved on the hard drive as a particular file. This file can be a prefab file, a Unity3d scene file, a text file containing code, a 3D mesh or any type of media as an image, sound or video file. Unity3D is capable of importing a huge range of

[1] The structure is used in the client application, in the big 3D monitor of the server application and in the Virtual Arm Settings window preview of the server application. The Hand State Editor inside the server application only uses the prosthesis without the arm, which is a prefab itself and included in the other prefab containing the virtual arm as well. Apparently it is also possible to create hierarchical structures using prefabs instead of GameObjects.

common media file formats and is capable of several coding languages as Javascript, C# and Boo to be used for developing components and several shader languages up to ShaderLab for developing shaders.

### 3.4.5 The Built-In Physx Engine

For physics simulations, as they are commonly used in 3D video games, Unity3D offers the built-in Physx engine. This engine provides features such as clothing simulations, the definition of forces and (partly) move-able joints, all of them provided as components, which can be attached to a GameObject. For the physics simulation which was created in this work, two further components are used, *Colliders* and *Rigidbodies*. These two components will be introduced in the following.

The *Rigidbody* component allows a GameObject to be influenced by the physics simulation. To be more precise, the Rigidbody component influences the Transform component of the GameObject it is attached to. Added to an empty GameObject, without any components than the required Transform component, the Rigidbody component would cause the GameObject to fall down - according to the gravity force defined in the settings. Usually, Rigidbody components should not be moved by using the Transform component since this would falsify the physics simulation. The required behavior for the virtual hand, being translated and rotated by the user, but still being able to interact with other physics-controlled Rigidbodies, can be achieved by setting the Rigidbody to the *kinematic* state. This stops any movement, caused by the physics simulation (even such caused through forces like gravity), and gives back full control over the GameObject's position and orientation to its Transform component. Another parameter, which was important in this work, is the type of collision detection to be used. For objects, which are moving around in a - at least for the physics engine - unpredictable way, like the prosthesis does, *Continuous* collision detection has to be chosen. For all GameObjects, which have a Rigidbody component attached to and are controlled completely by the physics simulation, *Discrete* detection is sufficient.

With this setup of components, the GameObject would fall through any other object because no collisions can be recognized. For being able to do this, first a "physical" body for the GameObject has to be defined. This is done by attaching one or several *Collider* components to the GameObject. A Collider component is available in several shapes, such as a box (cuboid), sphere or capsule (a cylinder with rounded ends). Furthermore a mesh Collider, which uses an attached mesh for defining the surface of the physical body of the GameObject, is provided. The control mesh for this last type of Collider is limited to 255 triangles, and the mesh has to be convex. Since it is possible to add several colliders of different types to refine the physical body for a Rigidbody component, in this work only the first three types of Colliders are used together for creating the required physical shapes. For creating immovable barriers like a floor, which keeps things from falling through it, an additional Rigidbody component is not required, unless the floor is intended to move around as well.

Besides its capability of causing collisions, the Collider component has a secondary functionality which also has been used in this work. If a Collider component is set as *Trigger*, it does not influence the physics simulation at all, but still generates three events: when entering an intersection with another collider object, while staying in the intersection, and when leaving

the intersection. This trigger functionality can be used, for example, to detect if an object enters, is inside or leaves a certain area, without influencing the physical behavior of this object.

### 3.4.6 About Networking in Unity3D

For supporting the creating of multiplayer games, Unity3D provides the built-in open source engine RakNet. RakNet's functionality is implemented in Unity3D by the class *Network*[1]. For being able to perform any data exchange among the components, first a connection has to be established. This has to be done the following way: RakNet provides a server/client structure with the idea of having several game clients running at the users computers, which all are connected to a game server. The server notices, when a user logs on or off, and furthermore can be used for managing the action happening in the game.

After a server has been started and a client established a connection to this server, it is possible to communicate directly from component to component. For example, the component, responsible for moving the fingers of the hand, can share this information with each other occurrence of itself in the network, ensuring that all occurences of hands in several applications are moving simultaneously. This is achieved by using remote procedure calls (RPCs), which allow to execute functions in another application on another computer via network.

The RPC implementation of RakNet in Unity3D allows to use several data types as parameters of RPC functions. These data types are integers, floating point numbers, strings and floating point vectors in $\mathbb{R}3$ and $\mathbb{R}4$ (quaternions) [29]. For the execution of a RPC function, the built-in *NetworkView* component is required, and has to be added to the sameGameObject, the component initiating the execution is attached to. A certain RPC function can be addressed by its name, if the other application has a NetworkView component with the same *NetworkViewID* than the one which is used for executing. Additionally, a component implementing the executed RPC function has to be attached to the same GameObject as the NetworkView component. basically, this is the procedure any communication between server and client application (components) is performed. Almost each component created for this work, which has to communicate over network, is able to update instances of itself. Each of these components can be marked "as sender" or "as receiver". Even both options are possible at the same time, causing the incoming update to be sent to further components. This can be useful, if one client has to share information with all other clients and the server. In such a scenario, the request would first been sent to the server and from there forwarded to all clients.

## 3.5 OpenTracker

The OpenTracker framework serves as an interface for transmitting tracking data in a highly modular approach. The process of transmission can range from just passing on data through serving as an interface between different communication channels[2] up to processing the incom-

[1] In this work the network functionality of Unity3D is only used for establishing data communications between the client and the server application. Data exchange from and to the Unity3D applications is performed through the OpenTracker framework.

[2] Like network and software protocols or even direct executing of function stacks.

ing data through several filters and modification modules. The focus of OpenTracker besides keeping high modularity and high performance in processing data was focused on the attempt to provide an easy, flexible and fast to change end user configuration, which can even be supported with graphical tools [26].

In this work, the OpenTracker framework is used for establishing data streams from the ioTracker Software into a Unity3D application for each tracking target. Furthermore, the framework provides the interface for the bidirectional communication between the EMG tracking device developed by Otto Bock, and the Unity3D application. The advantage of OpenTracker thereby is the possibility of splitting this process, which can be rather complex, into the part of exchanging data between the EMG tracking device and the OpenTracker framework on the one side, and exchanging data between the OpenTracker framework and the Unity3D application on the other side. Following this logic, for establishing an interface from ioTracker to Unity3D, it was also possible to make use of the already mentioned connection between OpenTracker and Unity3D after receiving data from ioTracker in the OpenTracker framework.

### 3.5.1 Modules, Data Sources, Data Sinks and Events

The task of sending and receiving data between a certain data source or sink and the OpenTracker framework is handled by a so-called *module*. Each module usually consists of one or more *data sources* and *data sinks*, but it is also possible for modules to just provide sinks or sources for unidirectional communication.

A data source creates an *event*, which usually contains a position, an orientation and a timestamp value as a minimum requirement of data needed for tracking a single ioTracker target. To each event, additional attributes can be attached, containing string, integer or floating point values. After an event was created by a source, it is pushed into the OpenTracker framework and moving through all specified modules, until it reaches the last data sink of the data flow. During its way through the OpenTracker framework, the data flow can pass several filters and modifiers, but it can also be split or merged. This allows a highly dynamical setup and easily swapping or reconfiguring of input- as well as output-devices or -applications, and is claimed to be a “write once, track anywhere” approach [26].

This approach of an setup easy to reconfigure is reached by using a XML file. Each data flow starts at a data source, described by a XML tag and related to an implemented OpenTracker data module (which are usually containing a sink and a source). For sending data to a special sink, the data source tag has to be inside a data sink tag. This sink tag again can be inside another sink tag, causing the data to be forwarded. At the beginning of the XML file, the configuration for each module used in the data flow can be specified.

In figure 3.5, the composition of a sample configuration file is illustrated. In the module configuration section at the beginning, a parameter for the Console module is defined. After the configuration section, a single data flow is defined. The data is pushed into the OpenTracker framework by a *VRPN* (Virtual Reality Private Network) *NetworkSource*, which is capable of receiving data, which was sent by the appendant *VRPN NetworkSink*. The generated data event afterwards is sent to the *EventTransform module*, which changes the scale and the rotation of the incoming tracking data. This is possible, because OpenTracker provides the mentioned above event data type, which clearly specifies a position and orientation value. After being scaled and

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE OpenTracker SYSTEM "opentracker.dtd">
<OpenTracker>
  <configuration>
    <ConsoleConfig display="on"/>
  </configuration>

  <ConsoleSink comment="Test OB Source" active="off">
    <UnitySink name="OBHand">
      <EventTransform scale="10.0 10.0 10.0" rotationtype="euler" rotation="0
      1.57079 3.14159" translation="0 0 0">
        <NetworkSource number="1" address="localhost" mode="unicast" port="12347"/>
      </EventTransform>
    </UnitySink>
  </ConsoleSink>

</OpenTracker>
```

**Figure 3.5:** Example of an OpenTracker configuration file.

rotated, the event reaches the *UnitySink* which causes the data to be pushed into the Unity3D application. The data flow in the OpenTracker framework is still not finished after this action, because the data yet has not reached the very last sink in the configuration. In this example, as illustrated in figure 3.5, the last sink is a *ConsoleSink*, causing the tracking data event to be displayed in the console window.

### 3.5.2 Adding modules to OpenTracker

As already mentioned, this work uses OpenTracker as an interface in order to receive the required tracking data, which is created by ioTracker and the OttoBock EMG tracking system, inside a Unity3D application. IoTracker was developed at the Technical University of Vienna and fortunately is compatible with the OpenTracker interface. This means, that the packages of data sent by ioTracker can directly be received by a VRPN Network Source. The connection to the EMG tracking device on the other hand is established through a serial port (COM), data is transferred by an individual protocol. These circumstances require an adequate process of reading data and pushing it into the OpenTracker framework, or - the other way round - receiving data from the OpenTracker framework and writing it into the serial port, keeping the specifications of the individual communication protocol.

In short, a OpenTracker module had to be implemented, which is acting as an interface between the EMG tracking device and OpenTracker. In the following, the process of creating a new module and adding it to the OpenTracker framework will be explained in general. For more information about the specific implementation of the described data-flows from ioTracker or the EMG tracker to the Unity3D applications above, please have a look at the chapters 5.3 and 5.4.

Each OpenTracker module class derives from the classes NodeFactory and Module (or ThreadModule, which itself derives from Module). In contrast to Module, ThreadModule provides functions for using a module-internal thread, for example, for receiving data through a

serial port. This class provides the following virtual functions, which then have to be overridden by the specific implementations of the respective module:

**Module**

- *void start()*: The start function is intended to start the module, right after initialization was done. When using this method from inside a ThreadModule, usually the the original Thread::start() function is called here.

- *void close()*: For closing the module and clearing all resources used by the module, this function can be reimplemented. Additionally, when used inside a ThreadModule, the original Thread::close() function has to be executed, which deletes the thread.

- *void init(StringTable& attributes, ConfigNode * localTree)*: For initialising the module if necessary, this function can be overridden. The parameters, which are passed with the function call, are defined in the module configuration section of the configuration XML file (see 3.5). As mentioned above, when used inside a ThreadModule class , the original *Thread:init(...)* function should be called at first, which then sets the initialisation flag for this module.

- *void pullEvent()*: This function can be implemented for pulling events out of the OpenTracker framework. It is called, after pushEvent was executed on any other module.

- *void pushEvent()*: For pushing events into the framework, this function can be implemented. However, in this work no use was made of the pushEvent() and pullEvent() functions, since the same functionality is provided by other functions, as *onEventGenerated*. More details about this approach are given after this itemization.

**ThreadModule**

- *void run()*: This function executes the internal receiver (and/or processing) loop in an individual thread. Such a thread is, for example, required for properly receiving data from a serial port (from outside the OpenTracker framework, and pushing it forward into the particular Data Source(s) of the module.

**NodeFactory**

- *Node * createNode( const std::string& name, const StringTable& attributes)*: This function is called for initialization as well, while the configuration XML file is processed. Each time, a data sink or source is defined in the configuration, this function is called, meant to create the respective sink or source class for the module. After the successful creation of a Sink or Source, this is passed as return parameter.

Additionally, the ThreadModule provides a *void lockLoop()* and *void unlockLoop()* function. These functions are not meant to be overridden, but used for safely getting data from-, or setting data to any variables used in the receiver loop. By locking the loop before any operation and

unlocking it afterwards, it is ensured that the receiver thread will not modify these variables in the meantime. In order to receive data at a data sink and forward it out of the OpenTracker framework, no receiver loop is needed. Instead each data sink provides an event handler, namely the function *void onEventGenerated()*, which is called each time an event was received. Similar to this, each source provides the function *void updateObservers(Event &data)* for triggering this event handler at all attached sinks.

As just mentioned, the module itself would be worthless without the implementation of data sources and sinks, as they provide the real interface to the OpenTracker framework by encapsulating the data into an event. Both, sinks as sources, derive from the class *Node*, which handles quite everything by itself - the only thing which has to be implemented, is the handling of pushing and pulling events.

### 3.5.3 The Unity3D Interface

Since OpenTracker is written in C++ and Unity3D C# is based on the .NET framework, an interface is required for gaining access to the C++ OpenTracker functions from inside Unity3D code. This interface is part of the *ARTiFICe* framework, which was originally created by the Interactive Media Systems Group at the Vienna University of Technology [21]. In short, the interface provides a C# wrapper for several OpenTracker classes, respectively functionalities. Most important in this work are the *Tracker* class encapsulating the OpenTracker UnitySink, and the *TrackingEvent* class providing the basic OpenTracker event functionality.

Basically, the interface provides the class *Tracking*, which allows to get access to a certain UnitySink by calling the *getTracker* method. The certain sink is identified by passing its name, as is was defined in the XML file, as parameter with the function call. The class returned by this getTracker method is the already mentioned above Tracker class and basically encapsulates the OpenTracker UnitySink class (C++) for accessing received OpenTracker events in Unity3d (C#) [21].

For this work the Tracking class was extended for providing access to UnitySources as well, especially for sending events. The UnitySource class had to be added to the OpenTracker framework as described in the chapter above. For sending an OpenTracker event through a certain class, the Tracking class was extended by the *generateEvent* function, which passes the name of the source and the tracking event as parameters and causes the passed event to be forwarded into the OpenTracker data flow. This extension enables bidirectional communicating, respectively sending and receiving OpenTracker Events from inside Unity3D. More information about the implementation of this extension can be found in chapter 5.4.

CHAPTER 4

# Interface and Application Design

This chapter discusses the creation of the virtual environment as well as the basic application design of the underlying software system. The client/server structure is introduced, which separates the system into two Unity3D applications. By introducing these applications, especially the server, which allows to define the process of the simulation, an overview on the scope of functionalities, provided by the virtual reality system, is given. Finally, the design of the grasping interaction will be presented.

Since the attempt was made to place the design of the user interface above the technical creation of the system architecture, in the following text, those two parts will be treated separately. In this chapter, the technical aspect of the implementation is left out to the extent that is possible, and will be presented in chapter 5.

## 4.1 Hardware Setup

The ioTracker system is set up in a room with eight cameras, which are aligned such that they can observe a space in the middle of the room with approximate dimensions of 4x4x3 meters. Inside this space, the protagonist is able to interact with the virtual environment. For visualization of the environment, the protagonist wears a head-mounted display (HMD), provided with a tracking target for determining the position and orientation of the virtual camera.

A HMD is usually built with two small LCD displays, one for each eye, and is therefore capable of stereoscopic viewing. This means, that each eye receives a slightly different picture, as it is true in the real world. The brain uses this difference, or binocular disparity, to “calculate” depth in the visual scene. The greater the displacement of an object when comparing these two pictures, the greater the distance of the object to the eyes. Stereoscopic viewing is a built-in functionality provided by the graphic adapter, and does not have to be implemented in the application. For sending both pictures, they are displayed in an alternating fashion.

As mentioned in chapter 3.3, the EMG tracking device provides wireless communication for sending controlling signals to the simulation. This setup allows for a much more natural and

authentic experience while interacting, since there are no issues of concerns regarding twisting (even oneself into) cables. For keeping this benefit, even when using a HMD, the protagonist wears a backpack, including a LiPo accumulator-powered supply for the HMD and for a wireless video (WHDI) receiver for retrieving video data for the HMD. More information about this setup can be found in [7].

**Figure 4.1:** The protagonist (right) is wearing a HMD attached with a tracking target. A second target is mounted to the forearm of the protagonist and translates this position to a virtual arm. The picture displayed in the HMD is also visualized by a video projector.

The control of the virtual hand, or more specifically, the control of opening and closing the hand is performed by EMG signals, as already mentioned. For controlling the position of the whole hand, which is attached to the forearm, many different approaches can be found in literature (see chapter 2.4.1). These reach from simply tracking the position of the hand (3-DOF), up to determining the bending of the elbow and the position of the shoulder as well. For this work it has turned out to be sufficient for a tracking target to be mounted at the forearm (or upper arm) of the protagonist, at the position where the stem of a prosthesis would begin. The position and orientation of this target (6-DOF), translated to a virtual arm and subsequently to the virtual hand, allows a natural experience of moving around.

In order to make the output of the HMD visible for other people to see, the video signal created by the computer is split and also sent to a video projector (see fig. 4.1). As explained in more detail in the following chapter, another screen (if necessary, on another computer) is provided for the monitoring and the controlling of the simulation process, and, for example, for guiding the training exercises.

## 4.2 Basic Application Design

As mentioned in the chapter before, for visualization of the virtual environment, the protagonist wears a head mounted display (HMD). Such a device is usually connected to the computer by the common DVI/HDMI display port and shows the same image that would be visible on a monitor connected to the display port. Therefore, the easiest way to create the image output for the head mounted device is an application, running in full screen mode and rendering the virtual environment through the perspective of the virtual camera in real-time.

Since common graphic adapters usually support the connection of two devices, for example monitors, it is appropriate to use the HMD as one device, while the other can be used for displaying an observation application for analyzing or guiding the action performed in the virtual environment on a monitor, for example, when performing exercises with a patient.

Following this idea, the decision was made to split the environment software up into two applications, one for creating the visual output for the HMD, and the other functioning as a controlling application. The decision of creating two applications instead of one, providing these two windows was made for several reasons. First, the free version of the game engine Unity3D, used for creating the virtual environment, restricts an application to one window. Another reason was the benefit of having two applications, capable of communicating with each other via a network. In the first subsection, the client/server structure will be presented in detail.

In the second subsection, a theoretical concept of extending this structure by one or more *spectators* is presented. Such an extension allows for the option of having more than one protagonist which is equipped to interact with the virtual environment and provides the option, for example, of creating collaborative exercising scenarios.

### 4.2.1 Client/Server Structure

As both the client and the server application are communicating to each other over the UDP network protocol, even if used on the same computer, the rate of synchronizing the received tracking data (at one of the applications) is limited. In order to reduce the latency of the virtual reality system to actions of the protagonist, the decision was made to retrieve and process incoming tracking data directly at the client application, which creates the visual output for the protagonist (see figure 4.2). Following this idea, and due to the requirement of creating bidirectional communication to the EMG tracking device, this whole process, including data transfer, also had to happen in the client application.

The built-in network engine of Unity3D called RakNet, is used in updating the position of the virtual arm and hand at the server application. The server manages the creation and destruction of the virtual objects used in the environment, and provides possibilities of changing parameters for several aspects of the simulation, most importantly mapping the created EMG controlling signals to a certain behavior of the prosthesis. If one of these parameters changes, this information is sent to the client. The client again processes the received tracking data for the purpose of positioning the virtual camera and arm, and according to the parameters defined by the server, the EMG signals are translated to move the prosthesis. All these motions are sent to the server, whereby the demand of providing them in real-time is not given.

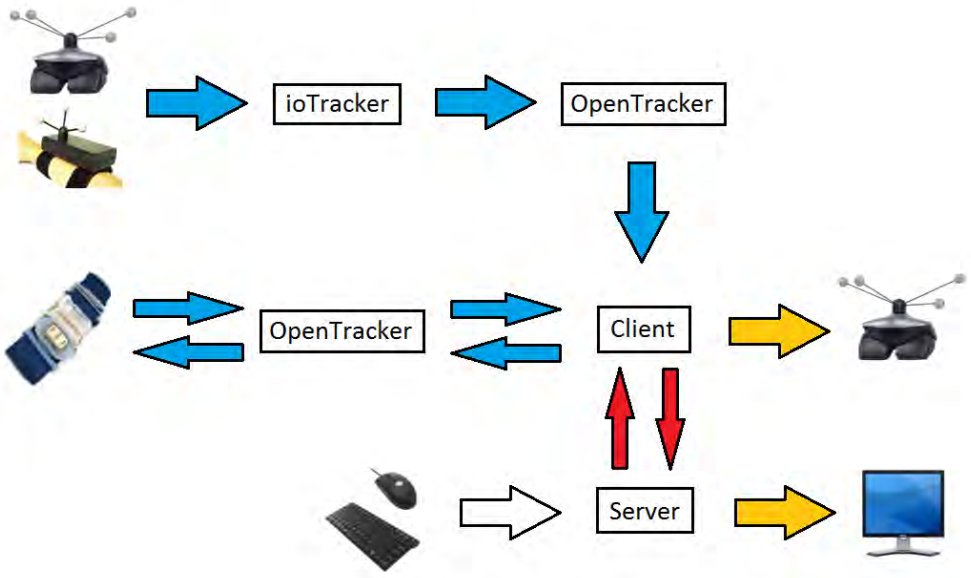



**Figure 4.2:** A work flow graph for the client/server structure. The blue arrows illustrate the data exchange between the client application and the tracking systems using OpenTracker, while the red arrows show the communication among the Unity3D Applications using the internal RakNet engine. The yellow arrows illustrate the video data generated for the HMD and a second display device.

This setup allows for the retention of the client application as purely an output, with (rather) no traditional GUI elements which are controlled by mouse and keyboard. Instead, the protagonist only interacts by using the virtual hand. All elements for interactions, which must be performed by mouse and keyboard, are placed at the server application.

For both the OpenTracker framework and also for the communication between the particular Unity3D applications, it is possible to exchange data over network. This allows for the encapsulation of the performance intensive task of tracking as performed by ioTracker to be run on a particular computer. This can also be done for the client application, which is no less important for achieving a minimal latency of the system as a whole. However, it is worth pointing out that, for this work, it was possible to run all applications on one computer, as mentioned at the beginning of this chapter.

### 4.2.2 Theoretical Spectator Extension

Since the client/server structure, provided by the built-in network engine of Unity3D, allows for the creation of multiple clients, it is appropriate to extend this structure by an additional VR spectator. This second virtual reality interface could be used, for example, to create “multiplayer” rehabilitation games, where exercises have to be performed in a team. In consideration of the fact that a protagonist’s motivation is one of the most important requirements for the success

of a rehabilitation game, such a “multiplayer” mode could help immensely. Another sample application would be to have an expert advising the protagonist from a first person perspective in reality. The experts arm could be optically tracked, enabling the expert, for example, to point at virtual objects. Watching the behavior of the protagonist from this perspective might also be beneficial in some situations.

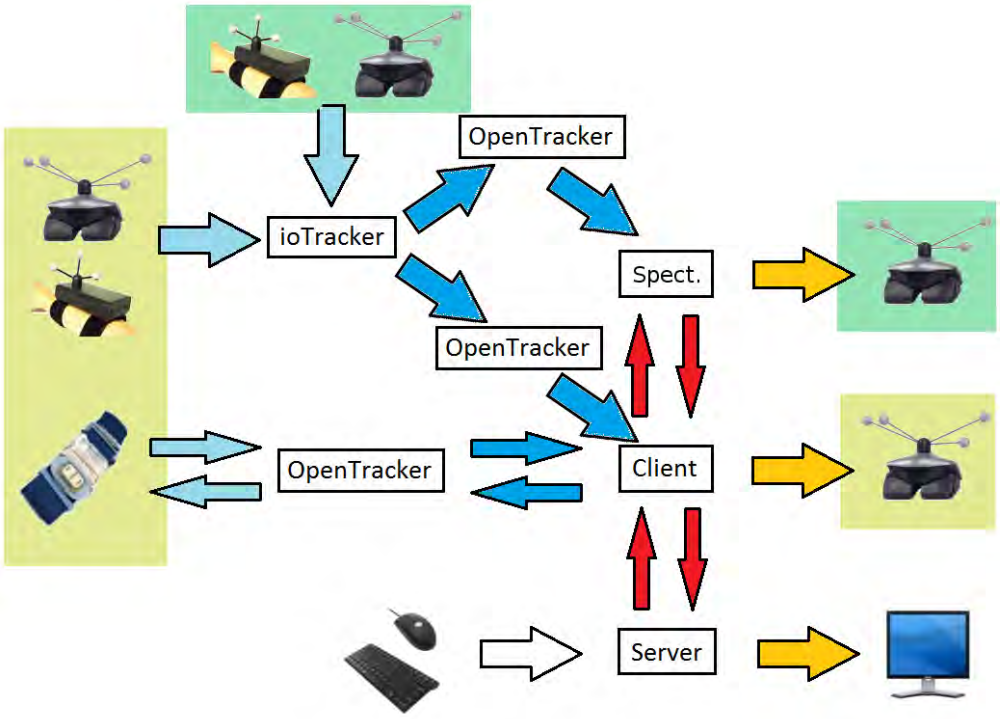



**Figure 4.3:** Spectator Extension work flow graph for the Client / Server structure. The blue arrows illustrate the data exchange between the client application and the tracking systems using OpenTracker, while the red arrows show the communication among the Unity3D Applications using the internal RakNet engine. The yellow arrows illustrate the video data generated for the HMD and a second display device.

For adding a second person to interact in VR, a second HMD, applied with a tracking target is required. A second arm tracking target would also be useful in providing a virtual pointing and interaction device. This task can easily be undertaken by ioTracker, since the use of up to 12 tracking targets is supported. A third Unity3D application would be needed to create the output image for the HMD, which directly receives the tracking data of the spectator’s head plus arm target positions from the OpenTracker framework, for keeping the minimal latency as well for the spectator’s output image (see figure 4.3). It is conceivable that a second protagonist could even be located remotedly, for example, at home (which often occurs in the context of rehabilitation). The provided RPC system (see chapter 3.4.6) for communication between the

server and the client(s) can easily be modified for distributing the head and arm position and the hand state of the prosthesis.

## 4.3 Client Interface - Performing the Interaction

As mentioned before, the client processes the incoming tracking data and generates an image of the virtual environment through the virtual camera controlled by the users head. For visualization, the model of an arm is controlled by the tracking target, which is mounted to the arm (stump) of the protagonist. For the virtual arm model, this coordinate origin can be defined by positioning a virtual equivalent of the tracking target in order to match the real position (see fig. 4.4). The bending of the elbow is not dynamically controllable by the protagonist, but can be set in the server application, as can the length of the lower arm, in order to perfectly match the real-life circumstances.

For the purpose of interacting with objects, the protagonist must move the virtual hand into a certain position over the desired object. In order to support the process of reaching the right position as well as, grasping, and releasing objects, some aides are available for the protagonist, such as the so-called object shadow, a grip force indicator and optical feedback for different parts of the grasping action. These aides, along with the process of grasping, are explained in detail in chapter 4.6.3.

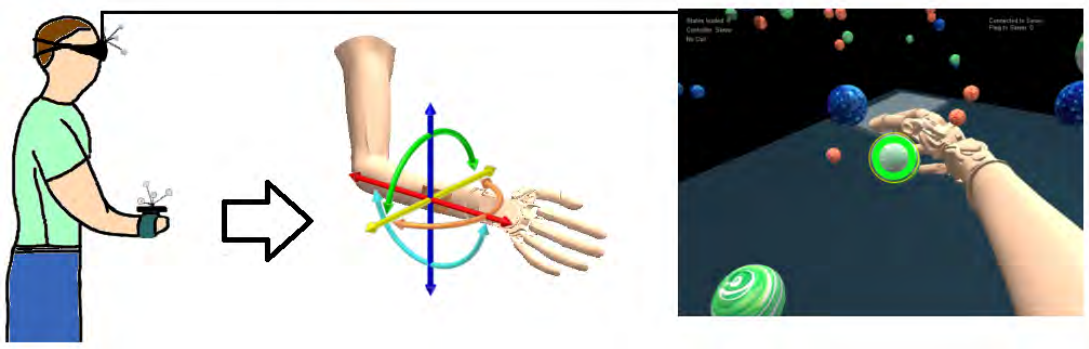


**Figure 4.4:** Left: A single 6-DOF tracking target is used to control a virtual arm plus the attached virtual hand. A second (6-DOF) tracking target is mounted to the HMD to control the virtual camera. Right: The image of the virtual environment created by the virtual camera, as it is displayed in the HMD. The bright green circle indicates to the user that the hand is in the right position for grasping the underlying object.

The graphical interface of the client application essentially consists of the image generated by the virtual camera. For additional information, a head-up-display (HUD) was implemented. In this work, the HUD only displays status information with regards to the connection status and several parameters set by the server. With a view to create a serious game, this HUD can easily be extended in order to display a score and similar tasks.

## 4.4 Server Interface - Controlling the Action

The server application is the back-end part of the virtual environment. All parameters concerning the progress of the simulation can be defined here. In the following subsections, an overview of all functionalities of the server application is provided.

### 4.4.1 Main Interface

The server interface window consists of a toolbar at the top of the window and a big 3D visualization of the training environment (see fig. 4.5). In this 3D view, the operator of the server application can view the actions performed by the protagonist through a camera independent of the protagonist. This camera can be translated easily by clicking and dragging the mouse wheel. For rotating the camera, the right mouse button must be clicked and dragged. The camera can be translated forward or backward along its view-axis by scrolling the mouse wheel.

In order to customize the simulation of grasping, the toolbar at the top of the application provides eight settings windows for different fields of functions. These functions range from starting the network server up to creating and performing exercises. In the following subsections, these eight control windows of the server application and the description of how to work with them will be provided. The implementation of the window system itself will be discussed in chapter 5.2.2.

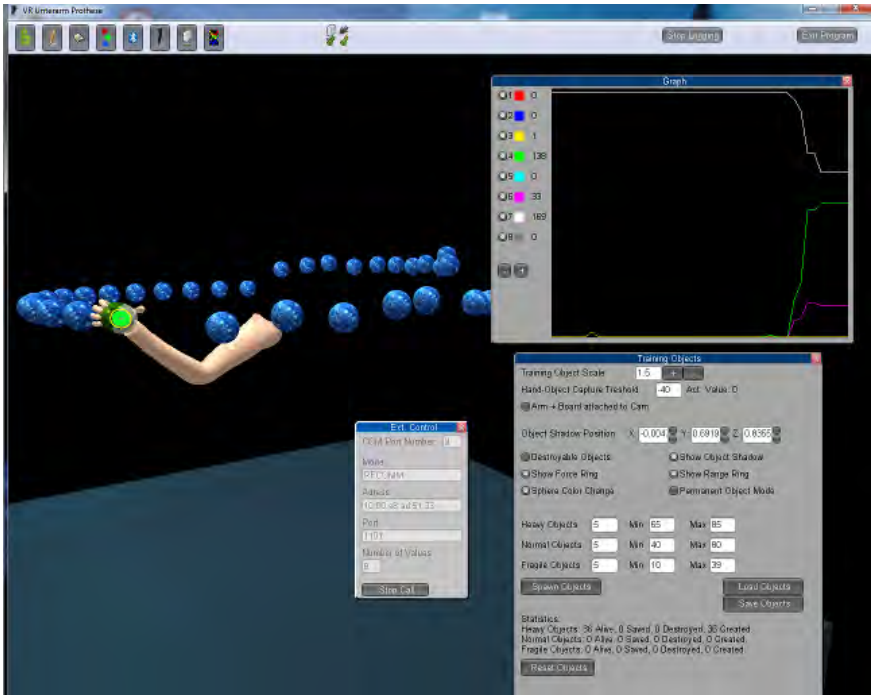



**Figure 4.5:** The server application in action, with several control windows opened.

Next to the buttons for the eight settings-windows, the toolbar contains a display of the network status, indicating if the network server was started and whether the client is connected or not. This gauge was implemented at an early stage of development due to unstable connections, and has proven to be useful in reminding the user that the network server of the server application has not been started.

Finally, even farther to the right is a button captioned 'Start Logging'. This button remotely starts and stops the process of data capturing at the client application. The *Data Logger* is used to record the actions performed by the protagonist in a CSV (comma-separated values) formatted file for later analysis. More details about the implementation of the data logger are given in chapter 5.7.3.

### 4.4.2 Mapping

The *Mapping Window* allows for the definition of a certain behavior of the virtual prosthesis for each incoming EMG control signal. The idea behind this concept was to provide a highly flexible system for setting up various different test scenarios. The concept of mappings and how to use them is explained in chapter 4.5.

### 4.4.3 Hand State Editor

Hand states are part of the mapping system and are used to move all or several parts of the prosthesis into a precisely defined position. This can be achieved by using them in combination with mappings [1]. The interface of the editor window essentially consists of a list of all the (move-able) parts of the prosthesis, and each part can either be set to "ignored" or set to a certain position. As illustrated in figure 4.6, the first four parts, namely the two axes of the wrist and the two axes of the thumb, are set to "ignore", while the other four parts, the four fingers, are all moved near their absolute minimum position, bent towards the palm. This hand state, used in a mapping, would have the following effect: The four fingers would move into the position, defined by the hand state, with the other four parts of the hand, the two axes of the thumb and of the wrist, are not moved at all.

For simplifying the creation of hand states, on the right half of the window the prosthesis illustrates the currently defined hand state. By left-clicking and dragging in the image box containing the prosthesis, it can be rotated around its center to be observed from any position. Additionally, by moving the mouse wheel, the camera can be moved towards or away from the prosthesis. Information about the implementation of the 3D preview window can be found in chapter 5.7.1.

In order to use the created states in hand mappings, it is possible to save the state with a certain name, which can then be found later in the mapping settings. Furthermore, it is possible to open and edit already created states. Since states are created as files in the sub-directory $server\_data/states/$ of the server application, they can simply be removed by deleting the file. The application should not be running when doing so.

[1] for more information about how to use hand states with mappings, see chapter 4.5

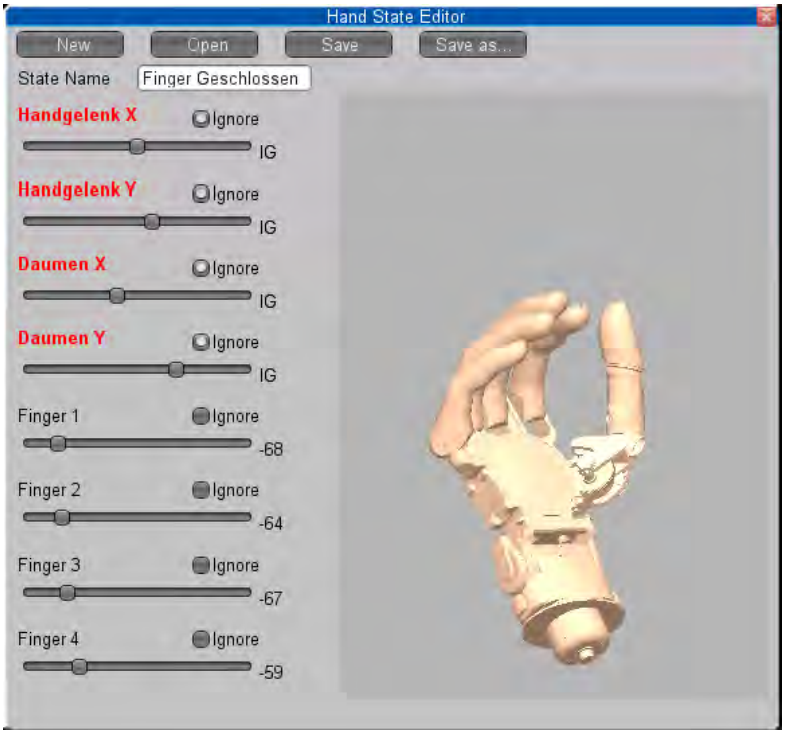



**Figure 4.6:** The Hand State Editor

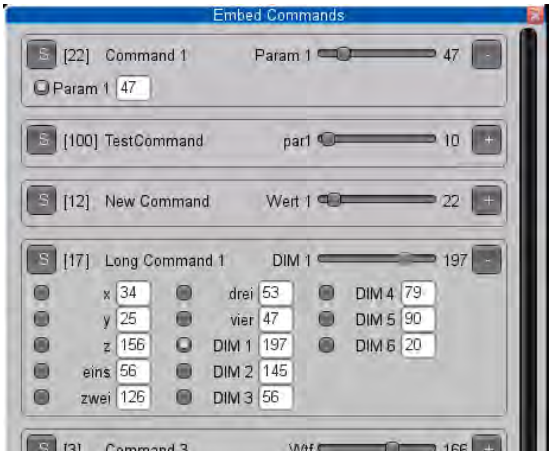
Due to a difficulty in creating the GUI window system, the hand state editor window closes all other windows when displaying a confirm box, if more than three windows are already opened. After replying to the confirm box, they are displayed again. This happens to prevent the stack order of the windows from being mixed up. More information about this technical problem can be found in chapter 5.2.2.

### 4.4.4 Training Objects & Test scenarios

This window provides controls for creating and destroying training objects, for adjusting the virtual environment and for loading and saving exercises or evaluation scenarios. Furthermore, the various grasping aides which are available can be controlled here. Operation of training objects and the process of setting up exercise scenarios are explained in detail in chapter 4.7.

### 4.4.5 Embedded Commands

Embedded Commands are meant to be used together with the Otto Bock EMG tracking device. These commands can be used to modify the processing of the raw EMG signals before they are sent to the OpenTracker framework. More Information about the implementation of the communication process is given in chapter 5.4.




```
<?xml version="1.0" encoding="UTF-8"?>
<COMMANDLIST>

  <COMMAND name="Command 1" id="22">
    <PARAM name="Param 1" value="47" />
    ...

  </COMMAND>

  ...
  ...

<COMMANDLIST/>
```

**Figure 4.7:** Left: The Embed Commands Window; Right: Example structure of the embed.xml file, defining the commands displayed in the window on the left.

The commands listed in the Embed Commands window are defined by a XML file, located in the data folder of the server application ($server\_data/embed.xml$). The schematic of the XML file is pictured in Figure 4.8: a COMMANDLIST tag includes several, but at least one COMMAND tag. The COMMAND tag consists of a name, an unique ID and one or more PARAM tags. A PARAM tag consists of a name and an integer value in the interval [0, 255].

When the application is loaded, a group of controls is created for each of the defined commands in the XML file. The parameters are listed with their predefined values to be changed, and additionally one parameter per command can be selected to be controlled via a slider. In order to send the command to the EMG tracker, the "S" button has to be clicked. When changing one value by sliding, as much changes per second as possible are sent to the EMG tracking device. This makes it much easier to test different values.

### 4.4.6 EMG Tracker Bluetooth Connection

This window is used for establishing a Bluetooth connection between the client application and the Otto Bock EMG tracking hardware. The window allows one to define the serial port provided by the Bluetooth receiver. An address, port and connection mode must be defined in order to connect to the EMG tracking device. Additionally, the number of incoming control values can be defined here, which basically is determined by the incoming data stream created by the tracking device.

After a connection has been established, the text fields are set to "disabled", and the button provides the option to close the connection again. After a connection to the EMG tracking device has been established, it is neccesary to stop the connection before closing the Application. Otherwise, the EMG tracker would still be in connection mode, which prevents any automatic reset or termination after the Bluetooth connection got lost. More Information about the implementation of this communication process is given in chapter 5.4.

### 4.4.7 Virtual Arm Settings

As mentioned in chapter 4.3, the virtual hand and the virtual arm are controlled by a single tracking target. Corresponding to a real prosthesis, which is mounted to a prosthesis stem, which again is fixed to the arm stump of the amputee, the virtual hand is mounted to the virtual arm. The reference point, on which the position of the real tracking target is translated to, functions as the virtual equivalent of the arm tracking target. It is visualized as a wooden plate with a purple point in the middle, modeled on the primal tracking target used for this application (see fig. 4.8).

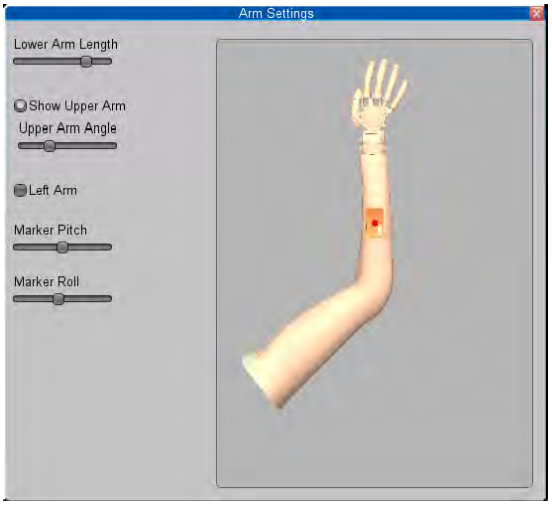



**Figure 4.8:** The Arm Settings Window displaying a virtual illustration of the tracking target. The target can be moved and rotated to be placed anywhere on the arm.

For best results in controlling the movements of the virtual prosthesis, the tracking target must be positioned as close to the hand as possible. Since each amputees arm stump is different in shape and length, the position of the tracking target for observing the movements of the arm differs for each amputee. This requires the option of modifying the position of the virtual tracking target in order to match the position of the real target for ensuring appropriate handling of the virtual prosthesis. To simplify this process, the virtual tracking target can be moved by left-clicking and dragging with the mouse. By right-clicking and dragging, the target can be rotated around its upwards axis. Rotating around the other two axes is done by the *Marker Pitch* and *Marker Roll* sliders.

For additional adaptation of the virtual arm to their respective circumstances, the length of the virtual forearm can be adjusted to fit the size of the protagonist. As mentioned earlier, it is not possible for the protagonist to bend the arm autonomously by creating EMG signals. Instead, this is just a cosmetic issue. When positioning the tracking target close to the end of the lower forearm or even onto the upper arm, the end of the 3D arm model can end up in front of the camera, occluding the scenery from the protagonist. To prevent this, it is possible to change the angle of the cubital joint. Alternatively, the upper arm can be completely hidden.

Finally, an option of using a left arm and hand versus the right ones is provided. Unfortunately it was not possible to implement this functionality as intended, which led to several bugs during the test phase. More information on the technical implementation of the virtual arm and on the difficulty of a left-handed prosthesis can be found in chapter 5.3.2. The implementation of the 3D preview window for defining the target's position is described in chapter 5.7.1.

### 4.4.8 Network Controls

It is necessary to start a network server at the server application in order to set up the simulation. Only then is the client able to find the server application and to request a connection. After the client has been connected, the HUD of the client application can be set to be displayed with full or little information, or can be turned off alltogether. Furthermore, this window allows one to define a certain port for the network address of the server. Finally, it provides a button for starting up and closing the network server functionality.

### 4.4.9 Control Value Monitor

This window visualizes the incoming EMG signals as they are sent from the client to the server application. Displaying can be turned on and off for each signal separately. With the "+" and "-" buttons, the temporal resolution can be increased and decreased. Since these values are sent from the client to the server with a sending rate lower than the one with which this signals arrive from ioTracker, it is not possible to display the full resolution of the signals as created by the protagonist. Information about the implementation of the line-chart window is provided in chapter 5.7.2. The EMG chart window was also used to perform certain user-tests with regards to practising the creation of appropriate EMG signals. More information about this test senario is provided in chapter 6.1.

## 4.5 Prosthesis Mapping

For the purpose of controlling the virtual prosthesis, the simulation makes use of the EMG signals created by the protagonist. Each muscle can affect one signal, which makes it nearly impossible to control a virtual hand in as many degrees of freedom (DOF) a real hand can naturally control. Actually, the number of DOFs which are provided by common prosthesis, is only two!

Therefore, the process of interacting with a prosthesis has to be simplified. The Michelangelo Hand, a prosthesis developed by Otto Bock, allows the amputee to use two controlling signals for moving the hand between different states (e.g. opened, closed), whereas one signal is used to move towards one certain state. Additionally, by creating both signals at once, the grip of the hand can be changed between two different states. In the following text, these states will also be mentioned as hand states.

While designing the application, one goal was to provide the option of experimenting and, therefore, there was a need in providing a high level of flexibility in how the prosthesis can be controlled by the incoming EMG signals. To achieve this, each incoming EMG signal can be combined (even more often than once) with a certain reaction of the hand. Such a combination

of an incoming signal and a reaction will be mentioned as *mapping* in the following chapters. The number of mappings, which can be used at the same time is unlimited and practically more or less limited to the number of incoming signals.

For combining a reaction of the hand to a specific incoming EMG signal with respect to the creation of a mapping, there are several modes available: three for defining a motion of the hand and three for affecting the actual grip force of the virtual prosthesis. Furthermore, it is also possible to simulate the grip force completely inside a motion mapping. Defining and editing mappings is done in the Mapping window of the server application. In the following text, all mapping modes will be explained in detail.

### 4.5.1 Hand Movement Mappings

#### Direct Mode

This mode allows the control of a single moveable part of the prosthesis. As the virtual prosthesis is based on the Otto Bock Michelangelo Hand (see fig. 4.9), the following parts are available for movement:

- Each of the four fingers can be moved as a whole
- The thumb can be moved in two dimensions, towards and away from the palm and up or down.
- The wrist can be moved in two dimensions. It can turn around the axis along the arm, and it can turn the hand up and down.

Each of these moveable parts can be assigned in the Direct Mode to move from one end of its boundary to the other, either by following the rising and lowering of the EMG signal or its inversion. For additional adaptation, the speed of the movement can modified by a multiplier. This multiplier is placed at the top of the window and called *Direct Speed* (see fig. 4.9).

#### Speed Mode

The functionality of this mode is very similar to the functionality of the Michelangelo hand prosthesis, as described in the introduction of this section. Each EMG signal is used to move the prosthesis into a certain hand state (e.g. opened hand, pinch, lateral grip). The higher the EMG signal, the faster the hand moves into the specific state. To prevent the hand from moving into several states at the same time in the event of several incoming EMG signals, only the mapping with the highest EMG signal is processed for Speed Mode. The hand states, which are used to define the state the hand should move into, can be created in the hand state editor, which will be explained in detail in chapter 4.4.3. As with the Direct Mode, an additional adaption of the movement speed is possible by modifying the appropriate multiplier. The control is placed at the top of the window, called *State Speed* (Figure 4.9).

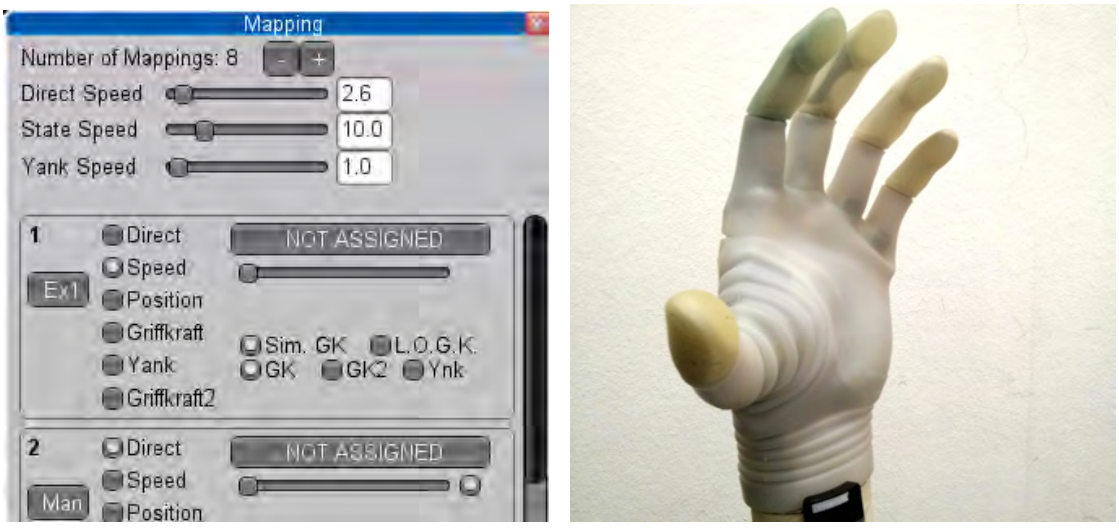



**Figure 4.9:** Left: The Mapping Window; Right: The Michelangelo hand prosthesis made by Otto Bock

#### Position Mode

Position Mode also makes use of hand states for moving the hand. In contrast to Speed Mode, only one EMG signal is used for controlling the movement between two specified hand states. If the signal is at its minimum (0), the prosthesis is in the first hand state. If the value is at its maximum (254), the prosthesis is in the second state. For each value in between, the prosthesis is moved into an interpolated hand state between the two specified ones according to the signal. Since inversion of this mapping can easily be done by swapping the first and second state, no inversion of the signal is needed in this mapping mode. Furthermore, it is not possible to modify the speed of the movement since each value between 0 and 254, as received from the EMG tracking device, is related to a position in the interval [0, 1] – interpolating between the two defined states.

### 4.5.2 Grip Force Mappings

In contrast to the previous mapping modes presented, the following modes are not used for moving the hand, but *only* for modifying the actual grip force of the virtual hand. Originally, grip force was introduced in the beginning of the second part of this work, and implemented as a result of creating closing signals while already holding an object. Then, the requirement was given to provide more flexibility when creating grip force. It should be possible to encapsulate the process of creating grip force from the grasping process, and to perform it outside of the simulation. The grip force should be defined by the use of controlling signals, which are received through the EMG tracking device (more information about the particular test scenario can be found in chapter 6.1).

This first led to the introduction of the grip force mappings and finally resulted in three different modes which will be presented in the following. Secondly, this led to the requirement of still being able to simulate grip force in a movement mapping. These simulation modes are presented in the following subsection (4.5.3).

#### Grip Force Position Mode (Griffkraft)

As the name of the mode already anticipates, this mapping mode modifies the grip force in a very similar way than it is done in the Position Mode. An incoming signal in the interval [0, 255] is mapped to the grip force interval of [0, 1], where zero means "no grip force at all", and one equals to the "highest grip force possible". Since there are no states to swap, like in the Position Mode, an invert option for this mode is useful to provide the option of mapping an incoming value of zero to the highest grip force as well.

#### Grip Force Speed Mode (Yank)

This mapping mode is similar to the Speed Mode. An incoming signal can be used for in- or decreasing the grip force, the higher the signal, the faster this happens. Therefore, to set up the actions of in- and decreasing the grip force, two signals (two mappings) are required, like it is also necessary for opening and closing the hand in the Speed Mode. The movement speed for adjusting the grip force can be modified with a multiplier as with the Direct Mode and the Speed Mode. This modifier is called *Yank Speed*.

#### One Way Grip Force Position Mode (Griffkraft2)

This mode can be best described as a mixture of the grip force Position- and Speed modes. This mode works like the Grip Force Position Mode, with the difference that one mapping can only increase OR decrease the grip force. The "direction" of the mapping can be set with the invert button. As two incoming signals are required in order to increase and decrease the grip force, this mode appears to be quite similar to the Grip Force Speed Mode(GFSM). However, in this mode setting the value for the grip force is still done in an absolute way, which means that the interval [0, 254] is mapped to the grip force interval of [0, 1]. In contrast to this mode, in the GFSM the actual value is in- or decreased by a certain amount according to the size of the incoming signal.

### 4.5.3 Simulating Grip Force

As mentioned in the previous subsection, simulating grip force originally was invariably combined with the mapping modes, intended for moving the prosthesis. With the implementation of the grip force mappings, additional requirements became necessary for keeping the flexibility of the mapping modes. As a first, this was the option of turning the grip force simulation on and off. Furthermore, the creation of several different grip force mapping modes, as mentioned in the previous subsection, made it necessary to provide the options for simulating all these modes as well. As a third point, the encapsulation of the grip force calculation from the actual events in the VR simulation created the requirement of using additional parameters in order to ensure

an accurate behavior of the hand movement mapping modes when calculating grip force outside of the simulation (see fig. 4.10). This problem regards *grip force mappings*, mentioned in the previous chapter, but due to its context, this issue will be explained in the following.


**Figure 4.10:** Options for simulating Grip Force at one mapping. Ex1 means, that the first one of the incoming EMG signals is used as an input.

Due to the specific implementation of the grasping process, using grip force is not possible in the Direct Mode. This is neither possible in combination with grip force mappings, nor by using simulated grip force. Without going into too much detail, it is not possible because in this mode no specific closing or opening action of the whole hand can be determined automatically, as only one finger or part of the hand is moved at once. More details about this issue can be found in chapter 4.6.3. In the following text, the options provided for simulating grip force and how to work with them in the Speed Mode or the Position Mode will be explained.

#### Speed Mode

In Speed Mode, the difficulty of simulating grip force reached its maximum complexity because here it is possible to simulate any of the three grip force modes available. Furthermore, when using a grip force mapping for creating the grip force externally, it is necessary to additionally block the according movement mapping. For this special case, the box “L.O.G.K.” (Lock On GriffKraft (german: grip force) has to be checked. In any other case, grip force is simulated. This can be done in the same three ways, grip force is also mapped from an incoming EMG signal (GK = Griffkraft, GK2 = Griffkraft2, YNK = Yank, look also at 4.5.2 Grip Force Mappings). Since in GK2 and YNK mode, one mapping can only be used to move the hand or grip force into one direction. Therefore, it is necessary to invert the simulated grip force at the one mapping, which controls the opening of the hand.

#### Position Mode

Position mode only allows to simulate grip force with the GK mode for a simple reason: If both opening and closing the hand are controlled by one mapping, it is also necessary to provide this functionality for in- and decreasing grip force. Since this is only possible in GK mode, the other modes would not work here.

## 4.6 Environment & Interaction Design

As mentioned in chapter 1.2, this work is based on two consecutive problem statements. As a first step, a virtual environment was created for grasping and moving virtual objects with a virtual prosthesis. In this first part, the main application structure and functionality was designed and a primal grasping environment was implemented. This first approach of implementing the action of grasping objects makes use of the built-in physics engine of Unity3D.

In the second part of the work, more specific requirements with regards to the environment and to the interaction were given. In contrast to the first specification, grip force should be used additionally when grasping objects. This addition required to re-design the process of grasping objects and new grip force sensitive training objects had to be developed. It was also necessary to create several grasping aides for the protagonist to visualize the amount of grip force. Furthermore, the demand was given to re-design the environment itself for a better virtual reality experience.

### 4.6.1 The Primal Interaction Environment

Initially, no specifications were given for creating the environment. The problem rather was to create the basic application structure, including the client/server applications and the extensions for the OpenTracker interface. In further, the hand mappings were developed and the first version of the interaction environment was designed. The environment, as illustrated in figure 4.11, was created with the idea of providing a comfortable, optically pleasing surrounding for exercising. It includes a stonelike plate, matching the dimensions of the tracking area, which is placed on a lawn. For easier orientation when moving the head around, the background was replaced by a skybox[1], which is providing an all-surrounding background image. The stone plate visualizes the actual area of interaction, which is limited by the tracking area. A table is placed on the stone plate, containing objects which can be grasped. However, besides of grasping these objects from the table and moving them around, no further options of creating exercises were provided.

The objects intended to being grasped were simple geometric bodies like cubes, cylinders and spheres, and could appear in different colors. For grasping an object, the protagonist had to move the opened hand towards an object and then manage to close the hand while keeping it over the object. Which sounds like the description of a very natural and daily situation has proven to be quite challenging when performed in virtual reality. For example, when grasping a cylinder, one had to take care of not pushing it away. This could easily happen by touching the object, even just with the tip of a finger, when moving the virtual hand towards it.

In order to create a realistic behavior for the interactive objects when being grasped, the built-in physics engine PhysX was used. The simulation provided by the physics engine allows to simulate gravity for objects as well as collisions with other objects in the scenery. The idea behind using the physics engine was to simulate the collisions between an object and the hand, for simulating a realistic behavior when the fingers start colliding with it. Objects like the table should not be influenced by the virtual hand at all. Unfortunately it turned out, that avoiding to

[1] A skybox is a huge cube, which is placed behind and outside of all other objects in the scenery. Its geometry is textured, for example, with images of the sky and creates the illusion of a far-away background.

**Figure 4.11:** Left: The environment design, as it was used in the first part of the work. The task was to simply grasp the objects on the table by closing the virtual hand around them, and move them around afterwards. Right: The training objects used in this first design were shaped as cylinders, spheres and cubes.

push the objects away was even more challenging than grasping itself, which should have been the actual exercise.

Furthermore, when the virtual hand was closed around an object, the protagonist had to take care in order to not lose it again while carrying it around. To avoid this problem, the object about to be grasped was "glued" to the virtual hand, after touching the index finger and the thumb of the virtual hand at the same time. Nevertheless, some problems could not be solved, as the objects sometimes were sticking inside the prosthesis palm without any option for the protagonist to release them again. More information about this difficulty can be found in chapter 5.6.1.

In summary, this approach of implementing the grasping interaction was not capable of creating a satisfying grasping experience, especially due errors resulting from the physics simulation. With regards to the actual goal of exercising the interaction of grasping objects, this approach was not sufficient.

### 4.6.2 The New Environment Design

Several circumstances led to the development of a new environment and interaction design. First of all, the use of grip force for the grasping process made it necessary to re-create the whole interaction. Furthermore, the problems which were experienced with the primal implementation of this process required to improve the interaction as well. Therefore, the decision was made to create a new, less realistic grasping process without the use of physics. The new grip force sensitive training objects allow to specify a certain interval of grip force for each. In order to be able to grasp and move around the particular object, the grip force created by the protagonist has to be inside the specified interval. If the grip force exceeds the boundaries of this interval during holding and moving an object, it gets lost and has to be grasped again. Additionally, it is possible to increase the difficulty by setting the training objects to "destructible ". This option causes the objects to "explode", if the grip force exceeds the upper boundary of their force intervals.

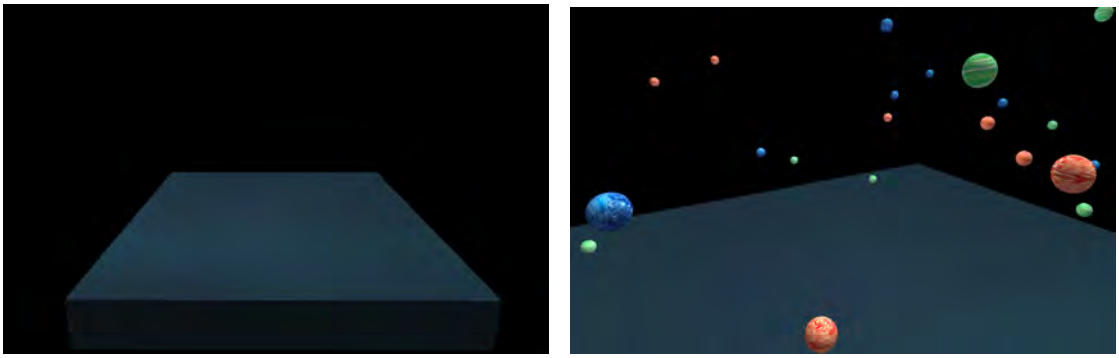


**Figure 4.12:** The second and final environment design.

In the new design, the shapes of the training objects have been reduced to spheres for a certain reason: When grasping a sphere, it is of no significance for the grasping action, from which direction the hand is moved towards the object, since the shape of a sphere is the same from any direction. Therefore, it is much easier for the protagonist to grasp an object, since the virtual hand does not have to be aligned additionally. As adopted from the first part of the work, the objects can appear in three different colors, namely red, green and blue. By default, each of the colors stands for a certain grip force interval.

Another specification for the second part of the work was to reduce the furnishing of the environment and to replace the sky by a black background. This finally resulted in a very minimalist design, only consisting of a dark green floor with the dimensions of the tracking space (see fig. 4.12). The idea behind this specification was to create an environment, which does not distract the protagonist. Gravity is neither used in this approach, basically to avoid a bad training experiences for the protagonist. Especially in the beginning of the rehabilitation process it can happen frequently that an object is not picked properly. As gravity would cause it to fall down to the floor in such a case, the protagonist would have to pick it up again, maybe even from the boundaries of the interaction area, it rolled to. Since the main task is about grabbing and releasing objects, gravity is not absolutely necessary for exercising it.

### 4.6.3 Grasping Interaction with Grip Force

In the new approach, due to the abandonment of the physics engine no collisions occur between the virtual hand and the training objects. This makes it difficult to decide, whether an object was grasped or not. Furthermore, the effect of shifting the object into the right position for holding it, as it was caused by these collisions, disappears completely. Therefore, two new conditions were introduced to provide a realistic grasping interaction. As first, the virtual hand has to be placed at a certain position relative to the training object for grasping it. This condition has the beneficial effect of the object already being at the “right” position when it is held, leading to a more realistic visualization of the grasping interaction. Furthermore, this leads to the solution for the second problem of detecting whether the fingers have already touched the object and should not move further. As mentioned in chapter 4.5.1, each finger can be rotated within a

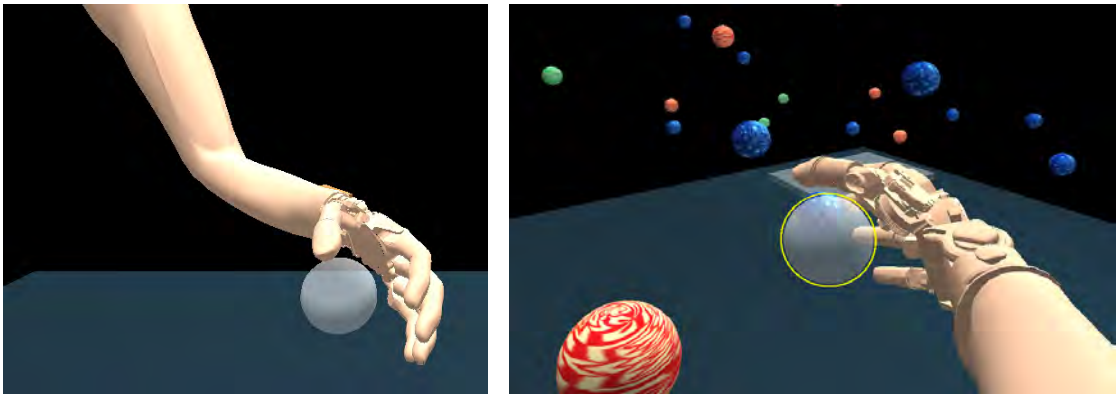


**Figure 4.13:** Left: The hand object shadow (grey sphere) grasping aide. Right: The hand object shadow in combination with the grip force indicator ring (yellow), which is indicating a grip force of 0%.

certain scope and each position thereby is defined by a certain angle. Depending on the hand state used, the index finger is at a certain position (angle) when the fingers start touching the surface of the object. The angle of the index finger is used as an indicator for this specific event.

Since the new approach requires a certain position of the hand relative to a training object for starting the grasping process, the interaction once again differs from reality and makes it much harder to grasp an object. This problem was solved by providing a positioning aide to the protagonist, the so-called *hand object shadow* (see fig. 4.13). This is nothing else than a semi-transparent sphere with the same dimensions as the training objects. The sphere is attached to the virtual hand and placed exactly at the same position, an object would have to be for being grasped. Since a single point in space would still be hard to find, the volume of a small cube collider is used instead for detecting, if this position was reached. More details about the implementation of the new grasping process are provided in chapter 5.6.2.

In order to manage applying the appropriate level of grip force when grasping a training object, the protagonist additionally is provided with grip force aides. Since grasping requires the eyes to be focused on the wanted object, these aides should not distract the protagonist's focus from the object during grasping. Therefore, the grip force indicator is placed directly over the virtual hand, with the idea of the hand being in focus when placed over an object. The force indicator is pictured as a yellow circle with the same dimensions as the hand object shadow, if no grip force is applied (see fig. 4.13). The higher the grip force, the smaller the circle gets. A similar representation is also used for the visualization of the grip force interval for an object. This interval is displayed as a thicker green circle, whose borders exactly match the limits of the force interval, according to the yellow grip force indicator. This means, that if the yellow circle is inside the green circle, the right amount of grip force was applied (see figure 4.14). Details about the implementation of these force indicator rings can be found in chapter 5.6.3.

The additional condition of applying the right amount of grip force when grasping an object caused the process of grasping to be divided into more acts than two, namely four: In the first act, no object is targeted. In contrast to the primal grasping process, in this act the prosthesis

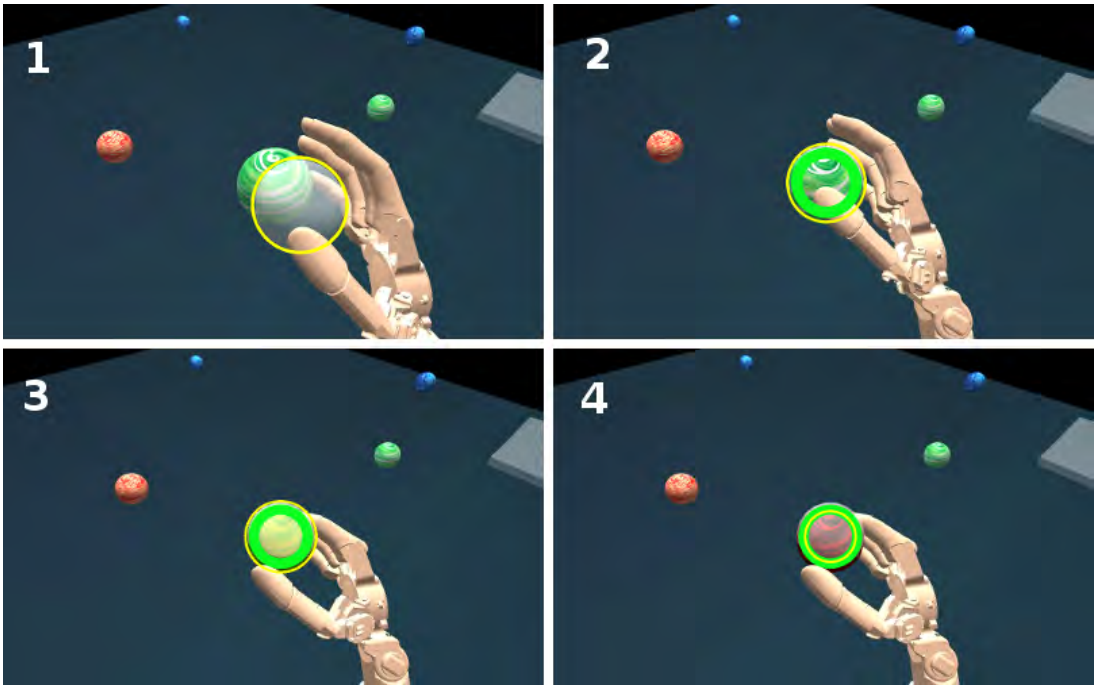



**Figure 4.14:** The process of grasping with grip force: (1) No object is targeted, force indicator (yellow ring) and hand object shadow (grey sphere) are displayed. (2) The hand is over the object, the grip force range of the object (10% - 40%) is displayed (green ring). Now grasping can begin. (3) The fingers touched the object and will not move further. The object turns yellow. (4) The grip force of the prosthesis (yellow) was matched to be inside the required force range. The object turns red. Now it is grasped and can be moved.

does not affect the training objects at all, even if it is moved through them. In the second act, an object was targeted, meaning that it is inside the required position for being grasped. This is communicated to the protagonist by displaying the grip force interval of the training object as a green circle.

Now the actual grasping action can begin. Since an object cannot be grasped while the fingers of the hand are closed, the grip force indicator will not be displayed in such a case. The opening of the hand is determined by the same threshold parameter, which is also used for determining whether an object was touched or not. If the fingers are closed as far that the threshold is exceeded, the third act occurs, indicating that the fingers have reached the object and will not close any further. Instead of moving the fingers, now grip force is applied (depending on the hand mappings used). The occurrence of this third act is communicated to the protagonist by turning the color of the object slightly into yellow. Additionally, this act is also indicated by the fingers stopping their movement, and by the grip force indicator, which starts decreasing its diameter. In this third act, the object is touched but not held. Therefore, moving the hand away from the object would not affect it at all and it would be lost again.

The fourth act of the grasping process can only be entered, if the amount of grip force applied is inside the boundaries of the force interval of the training object. Only under these conditions, the object can be held and in further moved around (see fig. 4.14). This event is signalized by turning the color of the object into red. If the grip force exceeds the boundaries of the grip force interval during this act, the object is lost. For re-grasping it, the hand has to be opened and then closed again. If the appropriate option was activated, the object will even explode if the grip force is higher than the maximum value specified by the training object. Additionally, there exists a fifth act, which occurs if a held object is moved into the target depositing area. This area is illustrated as a grey rectangle which is placed on the floor (see figure 4.14, in the background) and used for "rescuing" objects. This fifth act finally is communicated to the protagonist by turning the color of the object into green.

When releasing an object, the same acts as mentioned before occur in the inverse order. As releasing an object does require much less attention and accuracy than grasping, this part of the grasping interaction does not play a significant role. After the grip force exceeds the boundaries of the object's force interval, the object is released. Since no gravity is used for the new design of the environment, the object remains at the position in space where it has been released by the protagonist. The absence of gravity has proven to be very helpful in order to simplify the grasping process. Furthermore, it led to the creation of certain test scenarios which would not be possible with activated gravity. More information about using the target deposition area and about creating scenarios in general is given in the following chapter 4.7.

## 4.7 Preparing and Performing Test Scenarios

An important aspect of the virtual reality experience is the interaction with training objects. The server application provides a control window, which contains all options for adjusting this interaction. Furthermore, it is possible to prepare and perform training scenarios. In the following text, all options which are available in the *Training Objects* window of the server application are introduced, and examples are provided for how to work with these options. More information about the test scenarios performed with probands at the end of the work can be found in chapter 6.1.

### 4.7.1 Generating Training Objects and Test Scenarios

Basically, the whole process of performing (or preparing) training scenarios is all about generating training objects, which in further consequence have to be grasped and eventually placed somewhere else by the protagonist.

In order to generate training objects, two approaches are possible. First, objects can be spawned randomly by clicking the "Spawn Objects" button. This option allows to generate three types of objects with different grip force ranges at once. The number of objects for each type can be specified by the text-box next to the "Heavy"-, "Normal"- and "Fragile Objects" labels (see fig. 4.15). For each line, the "Min" and "Max" parameters specify the grip force range for the respective objects. Independent from the grip force range actually specified, the objects set at "Heavy Objects" are colored blue, the "Normal Objects" are colored red and the

"Fragile" ones are colored green. By clicking the "Spawn Objects" button, the newly created objects will be added to the ones already existing in the scenery. For each new object a free position in space is searched randomly. Such a position must be inside the tracking volume, not higher than 1.5 meters and no other object should already be placed in the close neighborhood. By clicking on the "Spawn Objects" button repeatedly, a huge amount of objects can easily be created.

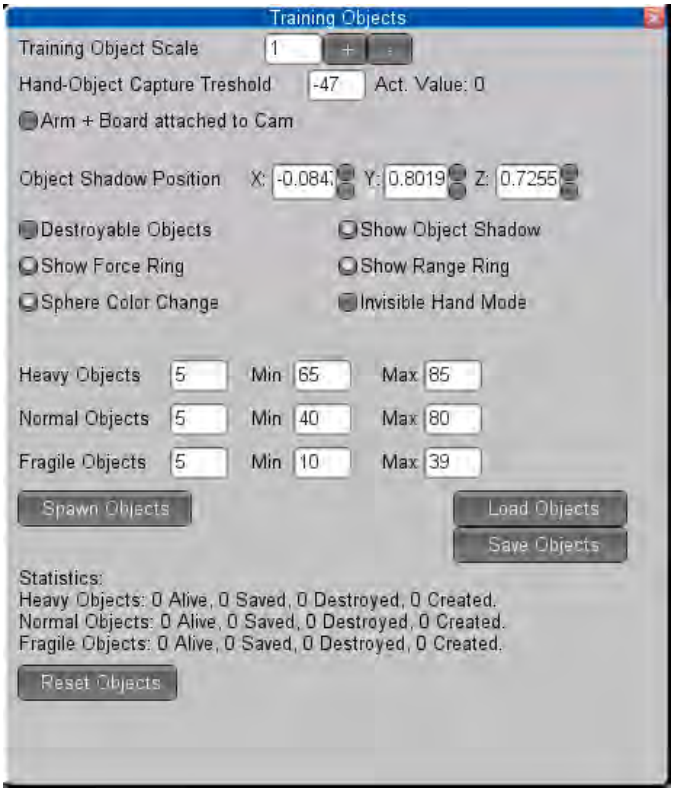



**Figure 4.15:** The Training Objects window, provided by the server application. This window allows to spawn and destroy objects, as well as to load and save arrangements of them. Furthermore the available grasping aides can be adjusted.

The second approach of generating objects is to load them by a file, which specifies the position, the grip force interval and the color for each object in the scenery. This file can be considered as a scenario itself and in contrast to the "Spawn Objects" button, all previous objects are removed when loading objects from a file. Therefore, this approach is useful when frequently restoring a certain testing scenario, for example, for the intense training of a certain movement or task. Creating these scenario files originally was intended to be done by spawning objects and then placing them by (the virtual) hand. After all objects are placed the right way, such a constellation can be saved by simply clicking the "Save Objects" button. The scenario information is always saved into the file *objectPosition.txt*, located in the same directory as the server application. In the same way, loading scenarios from this file can simply be achieved by

clicking the “Load Objects” button. In order to be able to work with different scenarios, it is necessary to rename or replace this file manually.

Instead of placing the objects by hand, it is also possible to edit the scenario file manually in a text editor. The format of this file is plain text and the parameters are set line-wise. The first line contains a number, which describes the count of objects specified in this scenario file. Then, for each object six lines are used to describe its attributes. The first line determines the color of the object, and can contain one of the three values “heavy”, “normal” or “fragile”. The second, the third and the fourth line describe the objects position on the X, Y and Z axis, and the fifth and sixth line determine the minimum and maximum boundary of the object’s grip force interval. These six lines are occuring repeatedly until all objects are described.

Removing a particular training object from the scenery is not possible. Instead, all training objects can be removed by clicking on the (“Reset Objects”) button. As mentioned before, this also happens when (before) loading a scenario.

### 4.7.2 Grasping and Depositing Scenarios

After the training objects have been created, the protagonist’s task is to grasp them correctly and place them somewhere else. Originally, the objects had to be placed in a certain target depositing area, however, it was not used in the tests that were performed at the end of this work (see chapter6.1). This target area is placed somewhere at the floor and ranges from the floor up to three meters. When moving a training object into this target area, the object turns green, indicating that this area has been reached. If the object is released, it will fall to the floor and disappear. This object now is counted as “saved”. Between the “Spawn Objects” and the “Reset Objects” buttons, an overview is given on the counts of objects generated, saved or still lying around in space(see fig. 4.15). The idea behind this overview was to provide testing scenarios, whereby the protagonist has to “save” objects in a certain time. To make this task harder for the protagonist, the objects can be set as “destroyable”, causing them to explode if the grip force exceeds the upper boundary of the object’s grip force range. These objects are displayed in the overview as “destroyed” objects.

Using the target board for depositing objects at a certain place requires the protagonist to head for the target area after grasping. Such a training scenario could be used, for example, to exercise the keeping of a certain level of grip force over some time. To avoid heading for the target area, a further mode was implemented: when activating the “Board attached to Cam” mode, the target board is always positioned relative to the virtual camera of the protagonist. The board’s position and orientation relative to the virtual camera can be set via the additionally displayed parameters “Board Position” and “Board Rotation”.

### 4.7.3 Permanent Object Mode

This mode provides a special scenario, whereby no training objects are needed. When this mode is activated, a training object is created and attached permanently to the hand object shadow aide. Therefore, no positioning of the virtual prosthesis has to be done and the protagonist is able to concentrate on creating the right amount of grip force for grasping the training object.

Additionally, when this mode is activated, two additional parameters appear in the *Training Objects* window in order to specify the grip force interval of the permanent training object.

### 4.7.4 Training Visualisation Settings

An important parameter, which has already been mentioned, is the "Hand-Object Capture Threshold". This parameter is used in the grasping interaction to determine, when (it looks like) the fingers start touching the training object (see chapter 4.6.3). This depends on the hand states used, on the position of the hand object shadow and on the size of the training objects. If any of these parameters is changed from its default settings, the threshold might have to be re-adjusted.

As just mentioned, it is possible to change the size of the training objects. In this work, it was necessary to adjust the size of these objects to fit a real object. This requirement arised in a special test scenario, in which the protagonist controls a real prosthesis by his actions in the VR simulation. This test scenario will be described in more detail in chapter 6.1.

Finally options are available for controlling the visualization of the grasping aides (described in 4.6.3). The force interval indicator and the actual force indicator can be hidden as well as the hand object shadow. Furthermore, the color changes of the training objects, which indicate the several acts of grasping, can be disabled.

CHAPTER 5

# Implementation

Since the previous chapter discussed the design-oriented aspect of this work, in this chapter the technical part will be discussed. In the first section, an overview of the underlying structure of GameObjects and classes is given, and the main parts of the structure are introduced. In the following sections, for each of these parts a more detailed description of the structure is presented, and an overview with regards to the implementation of the particular classes (components) is given.

## 5.1 Interplay of the Components

As introduced in chapter 3.4, the structure of a Unity3D application is based on GameObjects, placed in the scene tree. Each of these GameObjects can have components attached to it, which actually are classes and can interact with each other. These components allow to specify a certain behavior for the GameObjects, but also can perform background processes with no direct visual output. Following this idea, the structure of the applications is divided into several parts according to the structure of the GameObjects, which are containing the particular components.

Despite all attempts to illustrate the most important components and their relations to each other, it was not possible to include the hierarchical relation of the particular GameObjects without reducing the clarity. However, this is essential for understanding the functioning of the structure. As mentioned before, the whole structure can be reduced to several main aspects of the application (illustrated in fig. 5.1). This figure does not illustrate the complete structure, as several preview cameras and preview models (for example, for the hand state editor) also are placed at the root node of the scenery. These structures, which were used to create the preview objects, are similar to the parts of the structure that are mentioned above. Therefore, in view of the technical solution a complete overview of the implementation is given.

Referring to figure 5.1, the *Client / Server GUIObj* GameObject contains components for displaying the GUI (Graphical User Interface) and for the basic client/server network functionality. Issues with regards to the motion of the virtual arm as well as adapting the shape

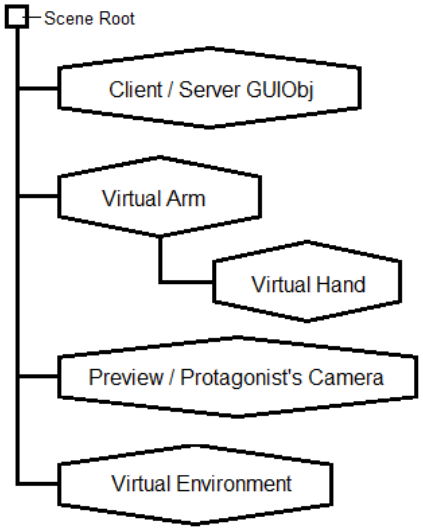



**Figure 5.1:** The structure of the main GameObjects, the application is based on. Each of these GameObject contains several other GameObjects and Components (classes), containing the implemented methods.

of the arm model or adjusting the virtual arm tracking target are represented by the *Virtual Arm* GameObject. The *Virtual Hand* GameObject contains the implementation for moving the hand model according to the mappings defined and the EMG signals created. Besides, the process of grasping training objects is partially controlled here. For creating the spectator preview at the server application as well as for the visual output at the protagonist's HMD, which is controlled by the incoming tracking data of the protagonist's head, the *Preview / Protagonist's camera* GameObject is the godparent. Finally, with regards to all aspects of creating training objects and performing test scenarios as well as the implementation of the grasping aides is represented by the *Virtual Environment* GameObject.

According to this structure, which is greatly simplified, the following sections of this chapter will present the particular parts of these structure with regards to their implementation.

## 5.2 The Graphical User Interface

In this chapter the technical aspect of using graphical user interface (GUI) elements like windows, buttons or text boxes in Unity3D will be introduced. Furthermore, and due to the difficulties caused by using the built-in GUI windows of Unity3D, the improved *GUIWindow* class with its appropriate *WindowManager* class - as used in the server application - will be presented and discussed. Finally, the *GUIObj* as mentioned in the previous chapter is introduced, which contains all the scripts for the GUI and for the (internal) RakNet network communication between client and server application.

### 5.2.1 Creating a GUI in Unity3D

Basically, creating a GUI in Unity3D is easy. Several control elements as labels, buttons, text boxes, sliders, check boxes and scroll-able areas are provided and easy to include. However, for special requirements, like drop-down menus, it has turned out to be quite hard to implement such a control element from scratch. In case of the drop-down menu, a workaround has been implemented, which is using buttons to rebuilt the menu. Examples for the use of this button-based menu can be found in the Mapping window of the server application. The difficulty of creating customized GUI elements lead to the decision to not use other elements than the ones originally provided by Unity3D, which of course restricted the design process of the interface significantly.

The implementation of the built-in GUI elements is done by calling them each time they have to be drawn. Each GUI component, which derives from the *MonoBehaviour* class, provides an *OnGUI* function, which is called every frame for drawing all GUI element, similar to its *Update* function. In the *OnGUI* function, each GUI element can be created by executing a function, which specifies the position and the default value of the element and returns the value entered by the user. For example, creating a text box is done by executing the following function *GUI.TextField*:

```
NewText = GUI.TextField(Rect(X, Y, Width, Height), OldText);
```

In a similar way, this can be done for all other elements that are provided. These functions are called each frame and just as the user enters a new value, it is returned by the function. Next to the "interactive" elements, which allow to change a value, there exists the box element, which allows to group other elements visually, and a window element, which is described in more detail in the following sub-chapter. It is possible to apply different "skins" to these GUI elements by using textures. Unfortunately the mapping of these textures onto their respective type of element cannot be changed and, for example, causes the blue title bar in each window used in the server application to be restricted in height. The feature of creating an own skin has only been used for adjusting the appearance of the window elements to provide a well-known design with a blue title bar, and for the text fields to provide a better contrast.

### 5.2.2 Server Application - GUIWindow System

Using windows in Unity3D initially seemed to be as easy as using any other GUI element. Problems started when displaying more than two windows at the same time, because the order of the windows, which are stacked one above another, gets mixed up each time a new window is opened or closed. This seems to be a problem in Unity3D 3.4 and is completely unacceptable for creating a pleasant user interface experience. The biggest problem in this context was the use of dialogs to open and save files, and confirm dialogs. The disorder of windows caused such dialogs to appear behind the window, which actually made them visible.

This difficulty caused the decision to improve the built-in GUI.Window element by implementing the abstract class *GUIWindow*. This class implements the execution of the original GUI.Window draw function and additionally contains event handlers for the window being

closed or focused. As this class is abstract, each of the control windows, which are implemented in the server application, are created as individual classes, all derivating from the GUIWindow class. The abstract function *drawContent* has to be redefined for each window to contain the definition of all GUI elements, as described in the previous subsection.

Additionally to the GUIWindow, the *WindowManager* class was implemented, taking care of drawing all opened windows in the correct order. Each derivation of GUIWindow, which gets created, has to be added to the WindowManager to ensure that the closing and focusing events are sent to the WindowManager. In order to draw all windows, it is only necessary to call the *DrawWindows* function of the WindowManager. In this function, the actual stack order of the windows is determined and then all visible windows are drawn in this order. Unfortunately - since for drawing still the original GUI.Window routine is used - the faulty ordering of windows still affects the new created windows. It was not possible to change the ordering effectively with the built-in functions *GUI.BringWindowToFront* and *GUI.BringWindowToBack*, since these functions seem to mix up the stack again. Anyhow, the focused window always gets drawn at the topmost position.

Furthermore, it was possible to solve the problem of file and confirm dialogs appearing behind the focused window, even if this solution might seem to be a little strange. Besides to the GUIWindows, the WindowManager class also contains the file and confirm dialog box windows. Since each GUIWindow has a pointer to these dialog boxes, the WindowManager is able to recognize, if one of these dialog boxes currently is set to visible. In such a case, only the topmost window (which has the focus and usually also caused the dialog box) and the dialog box window itself are drawn in the right stack order, since - as mentioned in the beginning - it is possible to define the order, if there are not more than two windows. After the dialog box is closed by the user, all windows are displayed again.

### 5.2.3 The GUIObj

The so-called GUIObj is simply a GameObject, which contains the component for rendering the graphical user interface. The GameObject itself is not visible, and has the only function of serving as a container for the components, which are attached to it. Since the interface for the client and the server application are different, two separate components are used for each, namely the *MainGui_Server* and the *MainGui_Actor*. For being able to communicate component-based, each application has to contain both components in its GUIObj, with the second one only needed as reference for the calling RPC functions and set to disabled[1]. Furthermore, the GUIObj contains all components, which are used by the control windows of the server and have no other GameObject to be attached to, like the component, which controls the communication to the Otto Bock EMG tracker (see fig. 5.2).

Additionally to the GUI components, each GUIObj, both at the server and the client application, also contains a component which is responsible for creating the basic client/server network communication, as it is provided by RakNet. This basic setup is not used for the communication between the particular components, but only implements the functionality of

[1] If a component is disabled, this simply means that it is not updated anymore and therefore does not receive any events such as *Update()* and *OnGui()*.

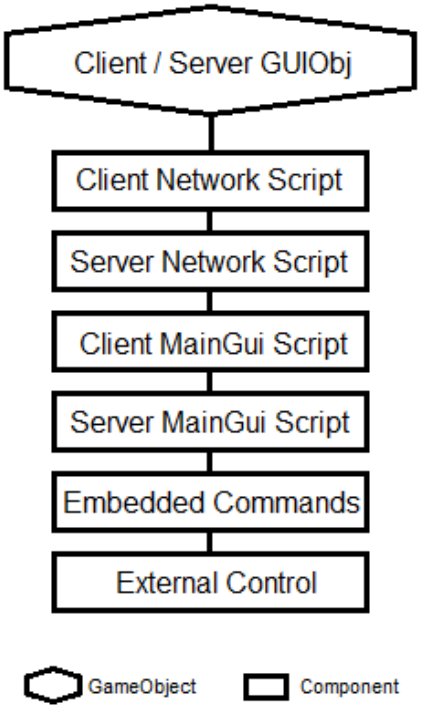



**Figure 5.2:** The Components attached to the GUIObj GameObject. Each GUIObj GameObject, at the server as well as at the client, contains several components of both applications for enabling direct communication from component to component.

creating a server and in further consequence connecting/disconnecting any client to it. As this is a basic functionality provided by RakNet, no special solutions were required.

### 5.2.4 The MainGui Components

As already mentioned before, for rendering the GUI two components exist, namely the MainGui_Server and the MainGui_Actor component. These components are not only responsible for rendering the GUI, but also have another important task.

If a client has connected to the server, the MainGui component of the server application calls the *remoteInitActor* RPC function at the clients MainGui component. When executing this function, the server passes the NetworkViewIds of all NetworkView components used to the client. This is necessary for the client application to "assign" each NetworkView component to one of the server application. After the NetworkViewIds are set properly, each component is able to send (and receive) remote procedure calls (RPC) to instances of itself occuring in one of the other clients, or at the server. After this process is finished, the client application is initialized and the components will start to exchange data with the server application.

## 5.3 The Virtual Arm

As introduced in chapter 4.4.7, the virtual arm acts as stem for the virtual prosthesis. Furthermore, it follows the movements of the real arm (stump) of the protagonist and therefore can be thought of as the "virtual equivalent" of the real arm (stump) as well.

Technically, the virtual arm consists of a main GameObject, which receives tracking data from ioTracker and moves itself according to this incoming data (see fig. 5.3). For visualizing the reference point of the arm, which functions as the origin of the virtual arm according to the position received by ioTracker, as well as for adjusting the shape of the virtual arm model, several more GameObjects are attached as children to the main GameObject and will be explained in the following. Finally, the virtual hand is attached to the virtual arm, since it has to move along with the arm as well.

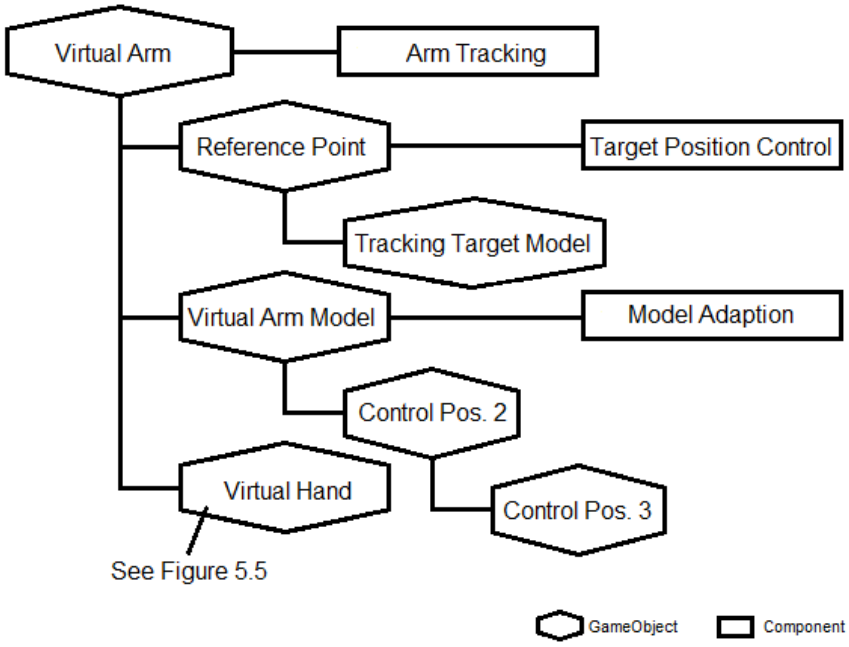



**Figure 5.3:** The GameObject structure of the virtual arm, including the reference point for tracking, the virtual tracking target, the 3D arm model and finally the virtual hand.

### 5.3.1 Receiving Tracking Data from ioTracker

Receiving tracking data from ioTracker inside a Unity3D application is a requirement of this work. Fortunately, ioTracker provides an interface to the OpenTracker framework and is able to send the incidental data as OpenTracker tracking events. As introduced in chapter 3.5.3, in further consequence these events are received by a UnitySink and made accessible from inside Unity3D by the UnityInterface. This functionality was already given and is not part of the work.

For processing the incoming data in Unity3D, the *HMDReceiver* component was implemented. This component makes use of the Tracking class, which is provided by the UnityInterface and listens to incoming events at the specified UnitySink. If an event was received, the received position and orientation are set to the Transform component of the main virtual arm GameObject, where the HMDReceiver component is attached to. For controlling the position and orientation of the virtual camera, the HMDReceiver component is used as well.

In order to share the virtual arm position with the server application, for visualizing the movements of the protagonist in the server applications preview window, the *Arm Position Sender* component was implemented. This component synchronizes all occurrences of itself (of the component) over network with the actual position and orientation of the virtual arm.

### 5.3.2 Customizing the Virtual Arm

As illustrated in figure 5.3, the GameObject which contains the model of the arm, is attached as a child to the main GameObject. This again updates its position according to the received tracking data. As presented in chapter 4.4.7, it is possible to specify the position of the virtual tracking target relative to the virtual arm for matching the position of the real tracking target. This is implemented by *not* moving the virtual tracking target, which always stays at the origin. Instead, the model of the virtual arm is moved and rotated in the opposite sense.

The ability of bending the forearm and hiding the upper arm was achieved by adding a skeletal bone model to the geometric 3D model of the arm. This feature can be saved in the 3DS (3D Studio) file format, and gets properly imported in Unity3D. The bones in Unity3D are visualized as empty GameObjects. By moving, rotating or scaling the Transform components of these GameObjects, in figure 5.3 simplified illustrated as *Control Pos. 2* and so on, the end points of the primal bone model are moved, causing several parts of the geometry to follow this movement more or less, depending on the distance to the particular bone. This, for example, allows to bend the arm by stretching and bending the skin instead of crushing the geometric model. As an additional benefit, adjusting the size and length of the individual parts of this single 3DS model can easily be performed by modifying the Transform component of the bone GameObjects.

## 5.4 An Interface for the Otto Bock EMG Tracker

In order to ensure a “realistic” behavior of the virtual prosthesis, this work makes use of the hardware developed by Otto Bock for controlling their prostheses. In contrast to the problem of receiving tracking data from ioTracker, no OpenTracker interface was available for this issue. Furthermore, communication with the Otto Bock EMG tracker device had to be implemented bidirectional. As mentioned in chapter 3.5.3, it was necessary to extend the functionality of the UnityInterface by the ability of sending OpenTracker events from inside Unity3D as well. Additionally, an Opentracker interface for the purpose of communicating with the EMG tracker device had to be implemented. The general approach of adding modules to the OpenTracker network was already presented in chapter 3.5.2. In the following, the focus lies on the implementation of an OpenTracker interface, capable of receiving EMG controlling signals and forwarding

them as OpenTracker events on the one hand, capable of receiving OpenTracker events and sending them as commands to the tracking device on the other hand. The implementation of this OpenTracker framework extension, as it is introduced in the following text, is written in C++ and has nothing to do with Unity3D.

### 5.4.1 Setup of the Connection

As introduced in chapter 3.3, the EMG tracker device is mounted to the arm (stump) of the protagonist and connected via Bluetooth to a wrapper module, which again provides access to the device through a serial port. For using the Bluetooth functionality of the wrapper module, ASCII text commands have to be sent through the serial port, using the iWRAP firmware protocol.

In order to connect the EMG tracker device to the WRAP module, first a connection to the wrapper module itself has to be established by opening a serial port. Then, the connection to the EMG tracking device has to be established by sending the appropriate commands through the virtual serial port. After this connection has been established, the EMG tracker immediately starts sending data back through the serial port, additionally the DTR (Data Terminal Ready) bit of the serial port is set to high. In this mode, it is possible to communicate directly with the tracking device for sending additional commands. For stopping the connection from the wrapper module to the EMG tracker, the DTR bit of the serial port has to be set to low for directly accessing the wrapper module again when sending data through the serial port, and the respective commands have to be sent. In the following subchapters, a closer look on these particular steps is given.

### 5.4.2 Establishing the Connection

When the serial port has been opened[1], a connection to the THOR module already has been established. For connecting the wrapper module to the tracking device, the following command has to be written to the serial port as text:

```
call 10:20:e8:e2:8d:33 1101 RFCOMM
```

The first parameter after the command *call* specifies the Bluetooth address of the EMG tracking device, the second parameter defines a channel and the third parameter the connection mode. All three parameters can be defined in the server application.

After this command has been sent to the THOR module, two responses are possible. The first line returned should always be “CALL 0”, indicating that the following line has something to do with the first call (connection) of the module, since more than one connection is possible at the same time. If the line returned would be “CALL 1”, this would indicate that another connection is still opened, which should not be the case in this scenario at all. If the connection has been established successfully, the second line returned is “CONNECT 0”, where the number zero refers to the number of the connection. Any other case signalizes, that the connection

[1]The functionality of opening and closing serial ports as well as sending and receiving bytes is provided by C++, and will not be explained in more detail.

could not have been established. Usually, this is indicated by the line “NO CARRIER 0”, but as implemented, any other value returned than “CONNECT 0” is interpreted as a failure.

After a connection has been established, the THOR wrapper module switches from command mode to data mode. In this mode, the DTR bit of the serial port is set to HIGH and the wrapper module is forwarding any data, which is sent through the virtual port to the device connected (in this case the EMG tracker). For stopping the call, the wrapper module directly has to be accessed, which is not possible in the data mode.

For switching the wrapper module back to command mode, the DTR bit of the serial port simply has to be set to LOW again. After this is done, the wrapper module can be accessed again directly by sending ASCII commands, which is signalized by returning the line “READY.”. For stopping the call, the command “CLOSE 0” has to be sent, were the zero once again refers to the number of the connection about to close. After the connection has been closed, either the line “NO CARRIER 0 ERROR 0” is returned, indicating that the connection was closed successfully. Another possibility is the return of the line “NO CARRIER 0”, which signalizes the occurence of an error, but nevertheless the connection was closed.

The greatest difficulty when implementing this process, was to prevent the system from moving into a deadlock state due to the loss of the Bluetooth connection. This happens, if the connection is aborted by external circumstances, and the devices do not react properly to this situation, like closing the connection unilaterally. So, for example, when the EMG tracker device “thinks” that the connection is still opened, it would not react to any attempts of recreating the connection, since the old one first has to be terminated. Since it was not possible to modify the EMG tracking device, this problem is still present. A similar problem was given by the THOR wrapper module, but as it is permanently connected to the computer and accessible via serial port, in this case it was much easier to “reset” the status of the connection in case of an error. This was implemented mainly by flushing the in- and output buffers of the serial port when starting a connection, and additionally sending a stop command to the THOR module for closing any still opened connections. This process is repeated three times, if problems are occurring when establishing a connection, and significantly reduced the need of fully reset the whole system (including a restart of the Unity3D applications).

### 5.4.3 Receiving Tracking Data

After the connection has been established successfully, the tracking device starts sending data. For each signal, the value sent is ranged in the interval [0, 254] and therefore can be sent as one byte. The value 255 is not used for directly transmitting signal data, but instead it functions as an end-of-line marker. For transmitting the values of the particular signals, all bytes describing the respective values are sent consecutively. After the bytes of all signals have been sent, the end-of-line marker - a byte with the value of 255 - is sent. Then the byte describing the first signal value is sent again, and so on.

For receiving, first the number of signals has to be specified. In this work, usually eight signals were transmitted due to the configuration of the EMG tracking device. Most of the time, only two of these eight signals were actually containing tracking data, and in further were processed by the Unity3D applications. For receiving eight signals, the interface tries to always read nine bytes at once, the eight signal values and the end-of-line marker. If the ninth bit is

not equally to the value of 255, this means that the values received are not corresponding to the respective signals. In such a case, the mode of receiving bytes is switched to only process one byte at a time, until the end-of-line marker was found again. After the marker was found, the receiving mode is switched back again to reading blocks of 9 bytes at once from the serial port.

Finally, the received values have to be put into a OpenTracker event in order to send them through the OpenTracker framework by a OpenTracker data source. Fortunately, besides the capability of holding position and orientation data, such an event allows to define additional parameters, furnished with a name and a value. By the use of this name, the value of each parameter can later be read from the event by any OpenTracker data sink.

### 5.4.4 Sending Embedded Commands

For sending embedded commands to the tracking device, for example, to modify the mapping of the raw EMG signal to a EMG control signal, first the according OpenTracker event has to be sent from the Unity3D application and to be received by the data sink of the OpenTracker EMG interface module.

As mentioned in chapter 4.4.5, such a command consists of a name, an ID and several parameters, each containing a value ranged in the interval of one byte ([0, 255]). When sending a command to the tracking device, the name of the command does not play a role, as it was only implemented for supporting the user in keeping an overview of the commands. The tracking device only requires the ID and the parameters of an command. Therefore, for sending the command to the device, it has to be prepared first. As it will be explained in more detail in the following subsection, the OpenTracker event received by Unity3D contains the ID of the command and the count of parameters. Additionally, each particular parameter value is stored. The names of all these parameters contained by the OpenTracker event are well known, which allows them to be read out and processed automatically. In order to send a command to the tracking device, it has to be put into the following format:

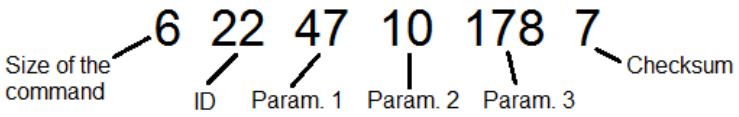



**Figure 5.4:** The structure of an embedded command, as it is sent to the tracking device. The command illustrated consists of six bytes, the checksum at the end is calculated as the sum of the previous bytes modulo 256.

The first byte of the command describes the absolute size of the command in bytes (including the checksum). The second byte sent is the value of the ID, which is necessary for the device to recognize the command which was sent. In the following, the values of all parameters are sent, each parameter thereby is described by one byte. Finally, as a last byte and also included in the size calculation for the first parameter, a checksum is sent. This checksum is calculated by summing up the values of all previous bytes sent, and then performing a modulo calculation

with the sum as dividend and a divisor of 256. This ensures that the value of the checksum can be represented by one byte as well.

After an embedded command was received successfully by the tracking device, this is signalized by interrupting the EMG signal data stream [1], and replying the whole command. Afterwards, sending the EMG signal data stream is continued immediately. Therefore, after an embedded command was sent to the tracking device, the process of receiving bytes and interpreting them as signal values is interrupted as well, until the appropriate command was returned or a timeout has been exceeded. Afterwards, the process of receiving bytes is continued as well. For ensuring a proper receiving without affecting the execution of the OpenTracker interface itself by, for example, waiting for the exceeding of timeouts, the process of receiving bytes from the serial port was encapsulated to be executed inside an individual thread.

### 5.4.5 Sending and Receiving in Unity3D

While the previous subchapters treated the issue of creating an interface between OpenTracker and the EMG tracking device, in this chapter the implementation with regards to Unity3D and OpenTracker will be discussed. As mentioned in chapter 3.5.3, an interface for this issue was already available and simply allows to receive OpenTracker events in Unity3D.

For the part of receiving EMG signals, as they are sent by the tracking device, this functionality was sufficient. The events, as received in Unity3D, provide the functionality of reading out all parameters, even the additionally added ones which contain the values of the EMG controlling signals. Therefore receiving the EMG signals can be performed in a quite similar way than receiving the tracking signals from ioTracker. The component *ExtControl*, attached to the GUIObj GameObject (as mentioned in chapter 5.2.3), takes care of the receiving and the storing of the actual value for each controlling signal. These values are then provided by the ExtControl component for further processing.

As already mentioned in chapter 3.5.3, in order to send events from Unity3D into the OpenTracker framework, the Unity3D interface as a part of the ARTiFICe framework had to be extended by an OpenTracker data source, which is capable of sending OpenTracker events. As such an Opentracker event was already implemented for the process of receiving, the implementation for the OpenTracker data source basically consists of creating a very common source with no special processing needed (for more information, please see chapter 3.5). The OpenTracker event for sending a command, as it is explained in the previous section, contains the ID of the command, the number of parameters used and the values of these parameters. Such an event is completely created in Unity3D and just looped through the Unity3D OpenTracker sink into the OpenTracker framework, where it is received by the OpenTracker source of the EMG tracker interface and processed in further.

[1] The whole process of sending embedded commands can only happen if the THOR wrapper module is in data mode and therefore the direct connection to the EMG tracking device is already established.

## 5.5 The Virtual Hand

As described in chapter 4.5.1, the virtual prosthesis used in this simulation is based on the Otto Bock Michelangelo Hand, especially with regards to its moveable parts. For creating a moveable 3D Model according to the prosthesis, each moveable part of the 3D prosthesis model first was separately saved into a 3DS (3D Studio file format) mesh file with its pivot point (origin of ordinates) aligned to the rotation axis of the moveable part. The rotation axis is the axis, such a part would rotate around when moving the prosthesis.

This has the advantage of such a mesh being easily imported into Unity3D. As the rotation axis of the mesh matches one of its coordinate axes, and its pivot point also is aligned with the rotation axis, the rotation of this moveable part can easily be accomplished by rotating the whole Game Object, which is containing the particular part as a mesh, around one axis. This is done by modifying the Transform component of the particular GameObject (see chapter 3.4.1). Furthermore, as the hierarchical structuring of GameObjects causes children GameObjects to follow the translations, rotations and changes in scale of the parent GameObject, it was obvious to create the moveable GameObject structure for the virtual hand according to the dependencies of the particular parts to each other. In figure 5.5, the resulting structure is illustrated.

This structure allows to control each part, which is intended to be moved, by the attached *Axis Controller* component. For processing the incoming controlling signals related to the mappings defined in the server application, further structuring was necessary.

Based on the idea of creating a hierarchical controlling structure, the moveable parts of the hand can be considered as the structure's lowest level. The second level may be formed by the *Hand Control* component, which has the task of assigning the moveable parts of the lowest level to the respective finger, thumb- or wrist-axis of the hand. The next level in this structure is formed by the hand states [1]. This level handles the position of one or more items from the lowest level to shape a certain hand state. Finally, the fourth and last level can be considered the processing stage for the incoming controlling signals created by the user.

For implementing this structure, basically each level is managed by a component. For controlling the rotation of a single move-able part around a certain axis, the *Axis Controller* component was implemented. Second, the *Hand Control* component takes care of accessing the respective controller for a move-able part and is capable of locking the fingers movements, for example, when touching an object. The *State Control* component as third level moves the prosthesis into a certain pre-defined hand state (accessing one or more Axis Controllers through the Hand Control) and the *Mapping Control* component translates the incoming EMG controlling signals created by the protagonist, depending on the mapping modes defined, into a certain movement, either of a single finger or thumb-(respectively wrist-)axis (by accessing Hand Control) or into a hand state (by accessing State Control) (see figure 5.6).

### 5.5.1 Axis Controller

The Axis Controller component basically is capable of rotating a Game Object around one axis. It is either possible to rotate the object continuously, or with a maximum and a minimum

[1] see chapter 4.4.3 Hand State Editor and 4.5.1 Hand Movement Mappings for more information

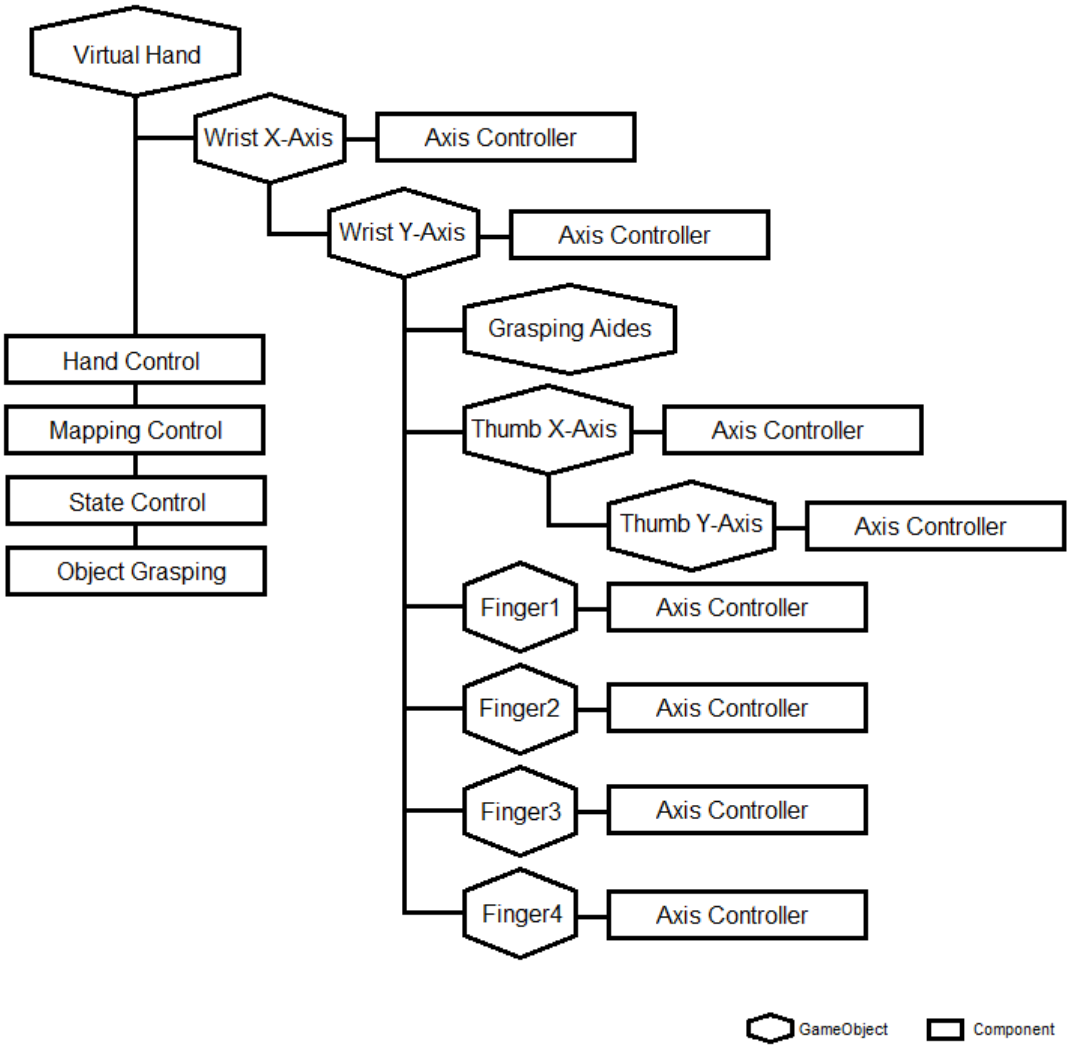



**Figure 5.5:** The GameObject structure of the virtual hand. Each moveable part (each GameObject) of the hand has an *Axis Controller* component attached to it for controlling the motion by the components attached to the virtual hand GameObject (Hand Control, Mapping Control...) .

boundary. Besides the task of keeping these boundaries, another important function of the Axis Controller is to provide the actual rotation value for other components. This rotation value is the angle of the object being rotated around the selected axis in degrees. As this value is passed on directly to the Transform component of the Game Object, it does not have to be in the interval [0, 360] but is converted automatically for rotating. Nevertheless, for providing an usable value for the other components when rotating continuously, it is kept in the interval [0, 360]. Furthermore, a default value for the rotation has to be specified, which should match the

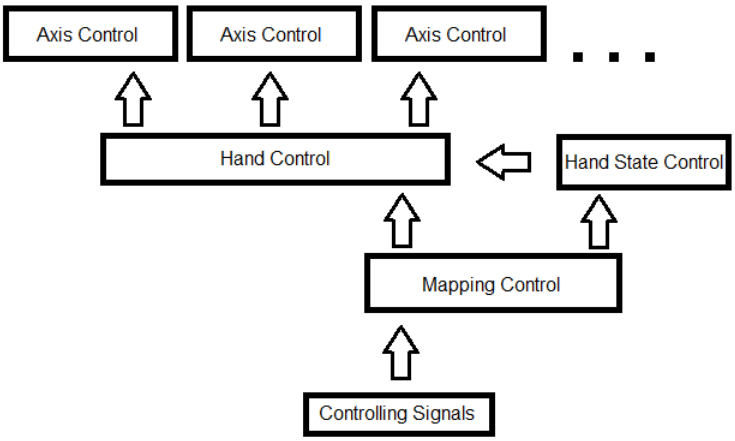



**Figure 5.6:** Signal data workflow for moving the virtual prosthesis.

default rotation angle of the respective part around its selected axis in the initial position. The Axis Controller component affects the game object, it is applied to, and automatically makes use of its Transform component.

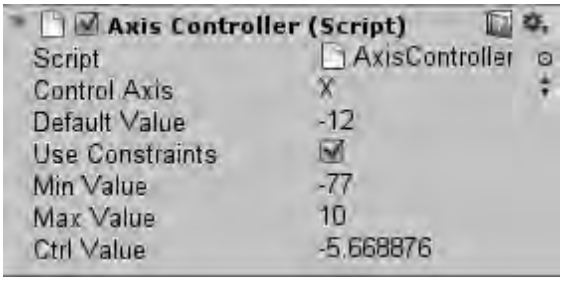



**Figure 5.7:** Parameters for the Axis Controller Component

### 5.5.2 Hand Controller

As already mentioned, the main task of the Hand Controller component is to provide an assignment of the eight Axis Controllers used to the respective parts of the prosthesis they belong to. In order to simplify this access for other components, each Axis Controller can be accessed through the Hand Controller directly via a specific integer index, or by using an enumeration data type for specifying the particular part of the hand.

Next to providing access to the particular Axis Controllers, two important values are stored and provided by this component. These are the actual grip force of the virtual hand, which is visualized by the yellow force indicator ring, and a threshold value[1] which is indicating, whether

[1]The rotation angle of the index finger

the fingers of the virtual hand have touched an object or not (see chapter 4.6.3). If the fingers are in contact with the object, the Hand Controller component can be set as locked, causing it to block the movement commands, as arriving from the higher levels of the hand's control structure, instead of forwarding them to the particular Axis controllers. In order to support the grasping process, two callback function pointers are available. These are executed by the ObjectGrasping component, when the fingers are opening or closing[1].

The Hand Controller component is able to synchronize all occurrences of itself over network by executing RPC functions (see 3.4.6). This means that the "original" virtual prosthesis, which is located in the client application and directly controlled by the actions of the protagonist, shares its actual state with all its other occurrences, like the virtual prosthesis in the server spectator view or an additional prosthesis in the theoretical spectator extension (see 4.2.2). For specifying the role of a Hand Controller, the component can be marked as "Sender", causing itself to share its state repeatedly. Instead of, or additionally to this, the component can also be marked as "Receiver", updating itself on each incoming RPC from the sender.

### 5.5.3 State Controller & Hand States

For managing hand states, the State Controller component makes use of the class *HandState*, which derives from the built-in Unity3D class *Object*. This is generally required in order to enable classes in Unity3D to reference to another class.

The main function of the HandState class is to store and provide all data necessary for describing a hand state created in the Hand State Editor (see chapter 4.4.3). This basically is the control value for each one of the individual moveable parts, and an unique name for the state. Furthermore, the HandState class provides functions for setting and reading these parameters in several ways, and furthermore the option of saving a hand state to-, or loading it from a specified file.

As mentioned earlier, the State Controller component is responsible for moving the prosthesis into a certain pre-defined position (the so-called hand state), by accessing the Hand Controller component and move each part towards its defined position. For keeping the flexibility, which is predetermined by the mapping user interface (see 4.5), an option is provided for moving the prosthesis towards a certain hand state by a certain amount (Speed Mapping). To ensure, that all moveable parts will reach their final positions at the same time, even with if some parts already close to it and others far away, the movement speed for each part is calculated individually. This is done by using the part with the greatest distance to its final posision as a reference, and according to this distance by speeding up the movements of all the other moveable parts.

A similar function, as used for the Position Mapping mode, is capable of setting the prosthesis into a hand state, which is created by interpolating between two given hand states. By the use of an additional parameter $w$ in the interval [0, 1] the weight for the first state is defined as $w$, while for the second state it is defined as 1 - $w$. Therefore, it is possible to move the hand from one hand state fluently into another. Since the parameter $w$ defines an absolute position for all move-able parts, the movement speed for each particular part does not have to be modified.

[1] Detailed information about the whole grasping process can be found in 5.6.2

Besides these two functions, another important task of the State Controller component is to function as a manager for all hand states, which are available in the simulation. As the hand states usually are saved at the server location, it is necessary to provide the client with all hand states at startup. Also, if a new state is created and saved in the editor, while the client application is running and connected to the server, this new state has to be transmitted to the client. This process of synchronization is done in a similar way as for the Hand Controller component, described in the chapter before, by defining a sender (which in this case is the server application) and a receiver component.

### 5.5.4 Mapping Control

The Mapping Control component as the topmost layer in the process of controlling the virtual hand, makes use of the Hand State component as well as of the Hand Controller component for applying the mappings as they are specified in the server application[1]. Due to the progressive adaption and extension of the functionality of mappings, especially of the several simulation modes for grip force, all the required parameters are stored in only six values.

Basically, for each mapping a control signal source has to be set and a mapping mode has to be defined. When using the Direct Mode, a moveable part has to be chosen, similar to selecting a Hand State when using the Speed Mode. In the Position Mode, two hand states have to be set in order to move the virtual hand between them. Furthermore, in the Direct Mode as in all Grip Force mapping modes, an option is provided for inverting the mapping. All these settings, five in number, are stored separately in an individual value. Additionally, for each mapping the value of the incoming EMG control signal from the selected signal source is stored, completing the six values as mentioned in the previous paragraph.

Basically, it is possible to store all required information by using the parameters listed above. However, for Speed and Position Mode the additional parameters are stored as bit flags in the parameter which was originally used for storing the index number of the second hand state, as required for the Position Mode. Assuming, that there will never be more than 256 hand states loaded and used, additional five parameters are stored in the five bits up from 256. Table 5.1 illustrates the meanings of these bit flags:

| Bit | Precondition | Meaning |
|---|---|---|
| 256 | Simulating: | Griffkraft |
| 512 | Simulating: | Griffkraft2 (GK2) |
| 1024 | Simulating: | Yank |
| 2048 | Simulating, Yank or GK2: | Invert Grip Force |
| 4096 | Not Simulating | Lock on Grip Force (L.O.G.K.) |

**Table 5.1:** The Bit flags used to store additional Grip Force simulation parameters in a Speed Mode mapping.

[1] A detailed description about *mappings* is given in chapter 4.5.

Besides its capability of processing the selected mappings, the Mapping Control component also handles the number of mappings used, and the multiplier options for modifying the speed of the Direct Mode and the Speed Mode, and in further the speed for in-/decreasing the grip force, as provided by the *Yank* Grip Force mapping.

All these parameters, the specified mappings as the values of the particular multipliers, can be shared with other occurrences of this component. This is required for the following reason: In the server application, the mappings are defined in the Mappings window and directly set to an instance of the Mapping Control component inside the server application. This mapping control shares all changes with another instance of the Mapping Control component, which is inside the client application, and responsible for moving the client's prosthesis. The movements performed by the client's prosthesis again are communicated to the server application by its Hand Controller component as mentioned in sub-chapter 5.5.2. The reason for this sophisticated procedure is once again to keep the latency of the VR system, especially the client application, as low as possible.

## 5.6 The Interaction Environment

Since the primal, physics engine based approach for an implementation of the grasping process turned out to be not satisfying for creating a realistic grasping interaction, a simpler approach was designed in the second part of the work. In chapter 4.6 an overview on this difficulty, and both approaches are introduced. This chapter will present and discuss the technical implementation of these two approaches. Furthermore, the remaining components used for the grasping interaction are introduced. These are the grasping aides for positioning and indicating grip force, the training objects themselves as well as their management for creating and destroying and, finally, the target object depositioning area.

### 5.6.1 Physx-powered Grasping

The physics engine powered approach makes use of some components provided by the built-in Physx engine, namely Rigidbodies and Colliders. A short introduction on using the Physx engine, and these components, is given in chapter 3.4.5.

Each Object, which is intended to be influenced by physics[1], has to contain a Rigidbody component. With regards to this work, such objects would be all the Training Objects and the virtual prosthesis, as they are intended to move around and interact with each other. The Training Objects should remain idle, until they receive any impact of another Rigidbody and therefore are completely controlled by the physics simulation. In contrast, the prosthesis is not intended to be affected at all when colliding with another object, but instead should be controlled completely by the protagonist's movements to match the real-life circumstances. As mentioned in chapter 3.4.5, for just building up an immovable barrier for Rigidbodies, using a Collider component is sufficient and no additional Rigidbody component is required. For the prosthesis, this would not

[1] Meaning that it is pushed away when receiving impacts of other objects, and falling to the floor, influenced by gravity.

work properly, since it is not only moving around fast, but furthermore doing so in a way, which is not predictable for the physics simulation at all.

Due to performance reasons, the Physx engine is not capable of handling those unpredictable movements sufficiently accurate, which causes Rigidbodies being able to move through a Collider in the worst case. For the special treatment, as it is required by the virtual prosthesis, the Physx Engine provides two essential parameters: By setting the Rigidbody to *Kinematic*, the Transform component of the GameObject is not influenced by the physics simulation anymore (see also 3.4.5). Second, the *Collision Detection Mode* has to be set to *Continuous*, which forces a faster update cycle for this Rigidbody Component to avoid the above described problem. Due to performance reasons, this mode should not be used with many Rigidbodies at the same time. Basically, this was no problem in this work, since the prosthesis is the only object which requires this special treatment.

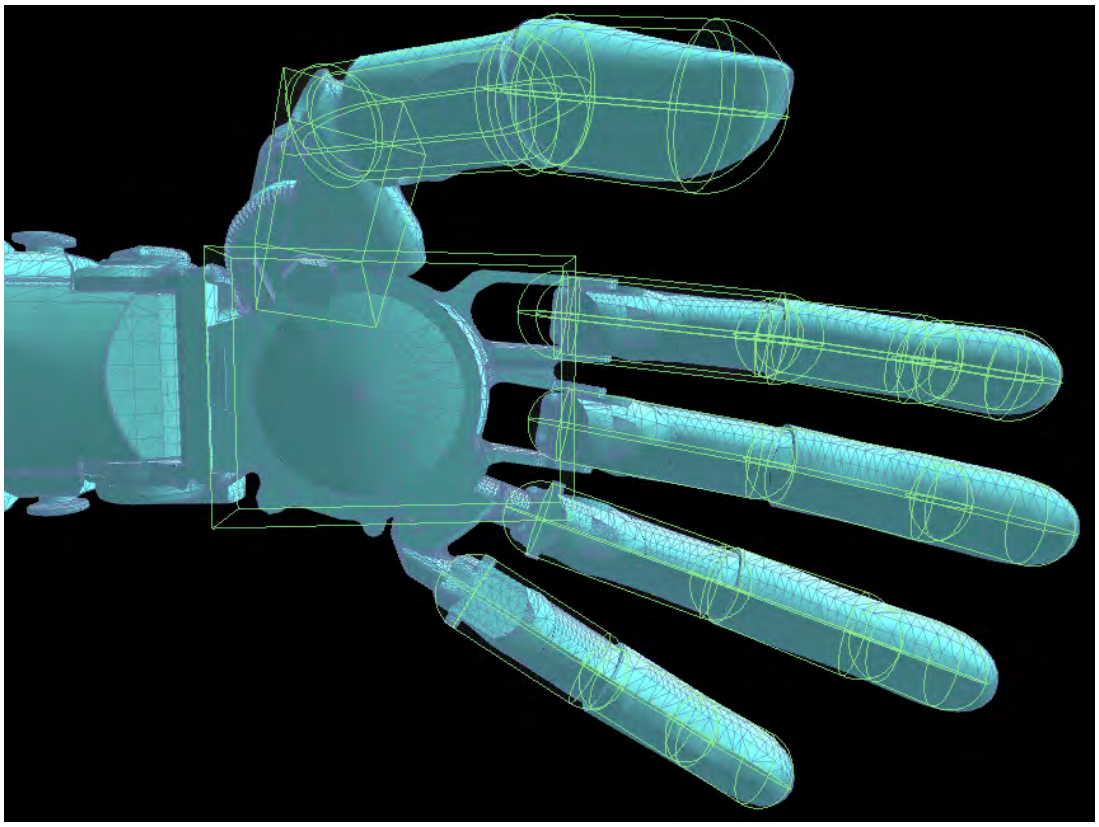


**Figure 5.8:** The virtual prosthesis with all its Collider Components (illustrated as thin green lines), as it was used for the Physx-powered grasping process. The thumb and the index finger contain two layers of Collider components, slightly differing in size and one placed inside the other. Covering the fingers is achieved by the use of capsule colliders, for the palm and the lower part of the thumb box colliders were used.

In order to add a “physical” shape to the Rigidbody component, colliders are used and attached to the same GameObject, which already contains the Rigidbody component. For the Training Objects, adding the physical shape was easy, since the primal training objects used in the first part as well as the new ones are shaped as cubes, spheres or cylinders. For these basic geometric shapes, colliders are provided. The most efficient way for shaping the physical shell of the hand was again to just use these basic shaped colliders. Since each GameObject can only contain one collider, but a RigidBody component handles all colliders from all GameObjects attached to it as children as well, each moveable part of the prosthesis - which is a GameObject itself - contains a collider (see figure 5.8). For eaxmple, for matching the bent shape of the fingers, three colliders per finger were used. This was achieved by adding empty GameObjects to the each finger, which are respectively containing the colliders.

After having set up all relevant objects with “physical” shapes, the main idea behind grasping them with the virtual hand was (due to the available grasp hand states provided by the Michelangelo Hand prosthesis), that any object can be treated as grabbed, when it is touched by the thumb and the index finger of the hand at the same time. As mentioned before, this event can be handled by using the *OnCollisionEnter* and *OnCollisionExit* functions, which are provided by each collider component. Additionally to catching these events, each collider of the hand, the ones not used for event handling as well, should ensure that no grabbed object can pass through the hand. In practice, this setup still caused problems like the hand moving through the objects instead of grabbing them or grabbing them too late, causing the fingers to stop inside the grabbed object. Furthermore, moving the arm still caused objects passing through the colliders of the virtual hand if performed too fast, and, for example, led to objects irreversibly sticking inside the palm of the hand.

For some reason, the OnCollisionEnter and OnCollisionExit functions did not work properly all the time. For improving the triggering of these functions, the *ArmSwitchDebug* component was implemented, which searches for a Rigidbody component and during runtime constantly changes the collision detection mode of this Rigidbody component between “Discrete” and “Continous”. This workaround seemed to help, but as the name of the component says, it was not intended to be the final solution. As mentioned in chapter 3.4.5, colliders can also be used as triggers, which disables their physical effect of creating collisions. The benefit of doing so was, that the *OnTriggerEnter* and *OnTriggerExit* functions seem to work more reliable than the collision event handling. Therefore, a second layer of colliders was built for the thumb and the index finger. This (outer) second layer should function as trigger, while the inner layer of colliders traditionally functions as the physical barrier for training objects (see figure 5.8). This setup did improve the reaction of the hand with regards to collisions with a training object, causing the object being fixed at the virtual hand until opening. The event of opening simply occurred in the *OnTriggerExit*, practically meaning that the thumb or the index finger moved away from each other again.

Even if this setup was working, it was not sufficient for being used as a virtual prosthesis simulation. The protagonist had to concentrate much on moving the hand over the object without pushing it away, and even if this was performed correctly, there was no guarantee that the grabbed object will behave correctly due to the collisions with the finger colliders. Instead, the collisions caused the grasped objects to stick irreversibly in the virtual hand or just being

tossed out of it several times. This difficulty led to a new concept of grasping, which will be introduced in the following chapters.

### 5.6.2 Simplified Grasping

The new approach for a grasping interaction, as it was developed in the second part of the work, especially adds the use of grip force to the grasping process (see chapter 4.6.3). Additionally, due to the problems which were epxerienced with the primal grasping approach, especially with regards to the physics engine, in this new approach the attempt was made to abstain from a realistic physics simulation at all. However, some functionalities of the built-in Physx engine are still used, as the recognition of collisions by using Collider components. Though, the visual part of the grasping interaction goes without the use of physics.

The main idea behind the new approach was to divide the grasping process into two steps. First, the virtual hand should be placed at the right position above the object about to be grabbed. This ensures, that the object is already at its final position for being held afterwards. Secondly, the virtual hand would have to be closed, until the right amount of grip force is reached. Then the grasping action is over, and the object is held. Actually, this process was divided into four "acts", which are described in detail in chapter 4.6.3). In the following text, the implementation of these acts will be presented.

The process of grabbing is controlled by the *ObjectCatching* component, which is attached to the virtual hand GameObject. This component receives an collision event, if the collider, which is placed at the center of the hand object shadow (see fig. 5.9), collides with a training object's own collider. This indicates to the ObjectCatching component, that the virtual hand is placed right above a training object, and initiates entering act two of the grasping process (displaying the green grip force indicator ring).

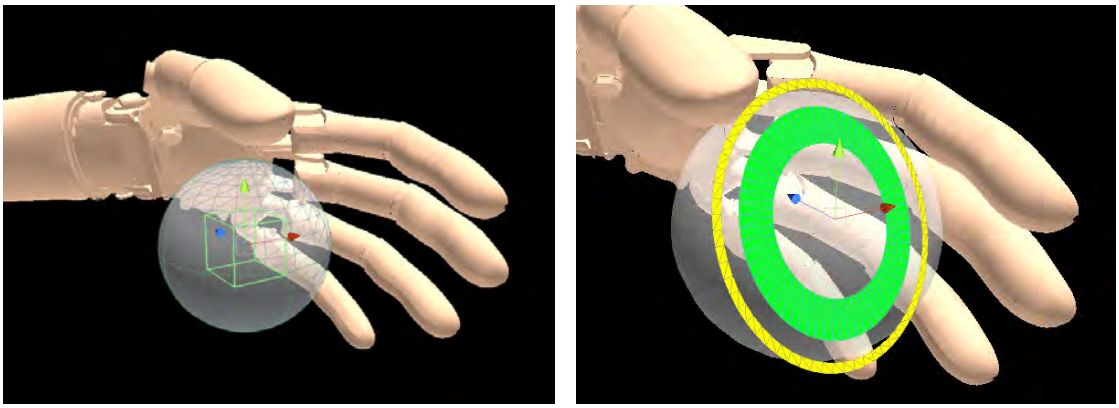


**Figure 5.9:** Left: The hand collision point collider, illustrated as green cube, is used for detecting, if the hand is correctly placed over a training object. Right: The two grip force ring indicators, in contrast to their usual appearence they are not facing the camera, and their geometry is visualized.

While the hand remains in the right position and is closing, the ObjectCatching component permanently checks the threshold for decide, when the fingers start touching the object. If the ObjectCatching component receives an *onCollisionExit* event, the hand was moved away from the object before holding it.

Assuming that the hand still remains in the right position, and the fingers already started touching the object, the third act was already entered as well. Now, the ObjectCatching component permanently checks the grip force, as provided by the HandControl component, to decide, whether an object is grasped hard enough for being held or even too hard, leading to the execution of the destroy method of this particular training object. If the grip force interval of the training object is reached, its GameObject is "mounted" as a child to the hand (according to fig. 5.5, it is attached to the *Wrist Y-Axis* GameObject).

While an object is held, the ObjectCatching component continuously checks the applied grip force, causing the object to be dropped again if the grip force drops under the required force for holding the respective Training Object. Technically, the GameObject of the training object is "dismounted" from the virtual hand by setting its parent to *null* (the scene root).

Additionally to the management of the grasping process, the ObjectCatching component also receives an event callback from the target deposition area, which is used for "rescuing" Training Objects and occurs if the protagonist moves the virtual hand into the space of the target deposition collider. If this event happens while the hand is holding a Training Object, reaching the "drop zone" is indicated to the user by turning the color of the object into green. If the object is released before leaving this area, the according method of the Training Object will be executed by the ObjectCatching component, causing it to fall down through the floor and finally being removed from the scenery.

The special function of a permanent Training Object, which is already sticking at the right position under the prosthesis, even if not held, is also implemented in the ObjectCatching component. This is easily done by creating and "mounting" a training object to the virtual hand, and thereby ignoring any incoming event callbacks.

Furthermore, the ObjectCatching component is taking care of displaying the various grasping aides like the range and force rings, the hand object shadow and the permanent Training Object, sharing the settings of how to display them over network as well. For synchronously displaying and hiding the range ring at the client and server applications (which indicates that the hand was moved into the right position over an object), this event also is shared by the client application.

### 5.6.3 Force Ring Indicators

As introduced in chapter 4.6.3, an essential aide for the grasping process are the two indicators for visualizing the grip force range of a training object and the actual grip force currently applied to the virtual hand. Each grip force indicator consists of a GameObject, that has an empty MeshFilter component (a container for model data) attatched, and further a MeshRenderer component for displaying the model data. Both components are built-in Unity3D components. Additionally, each indicator GameObject has attached the RenderTorus component, which creates the geometry of a torus mesh at initialization of the application, and passes the data to the MeshFilter component. Another way to achieve this would have been to create such a

mesh in a 3D modeling software and import it into Unity3D. The reason for creating the mesh by code was the easy modification of the inner and the outer radius of the range ring indicator.

```
Shader "RangeRingShader" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,0)
        _SpecColor ("Spec Color", Color) = (1,1,1,1)
        _Emission ("Emmisive Color", Color) = (0,0,0,0)
        _Shininess ("Shininess", Range (0.01, 1)) = 0.7
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }
    SubShader {
        Tags {"Queue" = "Overlay" }
        Pass {
            Material {
                Diffuse [_Color]
                Ambient [_Color]
                Shininess [_Shininess]
                Specular [_SpecColor]
                Emission [_Emission]
            }
            Lighting On
            SeparateSpecular On
            Blend SrcAlpha OneMinusSrcAlpha
            ZTest Always
            SetTexture [_MainTex] {
                Combine texture * primary DOUBLE, texture * primary
            }
        }
    }
}
```

**Figure 5.10:** The ring indicator shader written in ShaderLab language. Basically, this shader is a slight modification of the built-in Unity3D self-illuminating diffuse shader. Setting the “Queue” tag to “Overlay” and “ZTest” to “Always” ensure the wanted behavior. The properties listed in the first section of the code are accessible through the Unity3D editor interface, similar to the public parameters of a component (see chapter 3.4.2).

In order to create the torus, first two circles of points (vertexes) are created, additionally texture coordinates are created for each vertex. In a second step, faces are created by sequentially adding the indices of three vertices each time which describe a face. This process only happens once at startup of the application. For adjusting the size of the two ring indicators, the GameObject can easily be scaled by its Transform component. Each percentage can easily and exactly be visualized by defining the scale of one as a grip force of 100

A final requirement for the ring indicators was to display them always on top of everything else to prevent them being covered by the virtual hand or any training object. Additionally, they should always face the camera, which mathematically means that the direction from the camera to the middle of the torus should be perpendicular to the plane created by the faces of the torus. Since the position of the indicators should also be right above the virtual hand, respectively at the same point where the object is being grasped for easily keeping both in view, the easiest solution was to position the ring indicators at the center of the hand object shadow, which sloppy said is the spot the action takes place (see fig. 5.9). To encapsulate their functionality, they are children of the GameObject RingHolder, which contains a component with the same name. This

component receives the events for displaying and hiding the indicators as well as for adjusting their size. Additionally, the component takes care of adjusting the orientation of the RingHolder GameObject to always face the camera. This automatically leads to all indicators being oriented right, as they are children of the RingHolder GameObject.

In order to make sure, that the ring indicators are always painted on top, it was necessary to create a shader with disabled depth-buffer testing. In Unity3D, each geometry surface is described by materials, which again contain shaders. Even, for example, when painting a red object without any lighting effects, a shader would be defined. These shaders are coded in the ShaderLab language (fig. 5.10), which again can contain CG and GLSL shader code [29]. For achieving the required results, in this case it was sufficient to slightly modify the built-in self-illuminating diffuse shader by disabling the depth-buffer testing (setting "ZTest" to "Always") and additionally ensuring that this object will be put at the end of the render queue for drawing it at last (by setting the tag "Queue" to "Overlay", see fig. 5.10).

### 5.6.4 Training Objects

Training Objects are the objects intended for being grabbed. They can appear in different textures and contain a certain grip force range within they have to be grasped. Each Training Object consists of a main GameObject, containing the *Catchable Object* component, which holds the parameters described above. Additionally, each training object contains two children GameObjects, one containing a small collider positioned in the middle of the GameObject. This collider is used for creating the collision with the collider inside the hand object shadow for indicating, that the prosthesis is held in the right position for grasping. The other GameObject attached to the training object contains a particle system and a particle animation component.

The Catchable Object component provides methods for destroying (with or without showing the particle animation) and rescuing the object if released inside of the target area as well as callback functions for the *Object Manager*, which will be described in the following sub-chapter.

### 5.6.5 Object Manager

The main task of the Object Manager component is to spawn and destroy objects, and to share these events with the client application. The other way round, if the protagonist is interacting with a training object and, for example, destroying or saving it, this information is shared with the server. The moving of objects while they are held, as well as the interaction of grasping, and the adaption of the grasping aides to the grasping process are all calculated individually for the client and the server to reduce the complexity of the synchronization process. Only the important changes happen at the client and are then forwarded to the server, as saving or destroying a game object and the most important changes of the grasping process as touching, holding or releasing an object.

When spawning objects at the server, for each object spawned, a RPC method is called to create exactly the same object at exactly the same position at the client application. If a client connects to the server while training objects are existing in the scenery of the server, these objects are not spawned remotely. In such a case, the environment has to be reset and new objects have to be created. For managing the destruction (or rescue) of an training object, each training object

gets a reference to the Object Manager when it is created. By using this reference, the object is able to call certain methods as mentioned in the previous subsection, for signalizing the Object Manager that an object has been destroyed or saved.

### 5.6.6 Target Depositing Area

In order to extend the options of creating test scenarios, a target depositing area was implemented. This area technically seen consists of a collider, which is attached to a GameObject. When moving a training object into the space of the collider, an event for rescuing the training object as described in the previous subsection is triggered at the respective object. Additionally, the functionally was implemented to attach the target board to the camera for always being visible and reachable for the protagonist. This is performed by modifying the hierarchical structure of the GameObjects, and simply defining the virtual camera as new parent for the board. Adjusting the position of the board can be done in the training object settings of the server application.

## 5.7 Further Implementations

In this chapter, the concept of creating 3D previews will be presented, as it is used, for example, in the Hand State Editor window or the Arm Settings window of the server application. Furthermore, the implementation of the EMG line chart window, as used in the server application as well, will be presented and, finally, the implementation of the DataLogger component will be discussed, which is used for recording data of the simulation process for later analyzing.

### 5.7.1 Preview Cameras

For creating 3D preview windows, as they are used in the Handstate Editor- and Arm Settings windows of the server application (see chapter 4.4.3 and 4.4.7), an individual camera component is required for each preview. The model of the hand or the arm, which should be visualized in the preview, also has to be an individual GameObject in order to be controllable independently from the ones, wich are used for the main preview camera of the server application (which displays the current action going on in the virtual environment of the client application). Since there is only one Unity3D scenery for putting all these objects together, it is necessary to separate them in a way that each camera only displays a certain selection the GameObjects. This can be achieved by using the built-in functionality of layers.

Usually a camera displays all GameObjects, independently from the layer they are assigned to. By setting the *Culling Mask* property of a camera to a certain layer, this camera gets restricted to only display GameObjects which are assigned to the respective layer. In this work, for each preview camera an individual layer was created. In order to control the preview models of the arm or the hand, the same components are used as for controlling the objects in the simulation itself. The only difference is that they neither send nor receive any status information over a network. By using the same components for the preview as well, it is guaranteed that the previews match the final results in the simulation.

In Unity3D, the image of the virtual environment created by a camera is not visible, except if the camera was selected as the main camera of the scenery - then the picture of the camera is rendered into the main view-port (the window) automatically. For rendering an additional camera into a preview window, first the GUI has to be drawn. Then, the view-port bounds of this camera are limited to the exact position and dimensions of the desired preview rectangle area. This can easily be done by setting the *pixelRect* property of a camera to the desired area. Afterwards, the camera image can easily be rendered by executing the *Render()* method of the camera. This has to be done every frame after the GUI elements have been drawn, and therefore is executed in the *OnGUI* method of the GUIObj GameObject.

### 5.7.2 EMG Line Chart

As introduced in chapter 4.4.9, the server application provides a line chart visualization for the incoming control (EMG) values. The rendering of the chart lines also requires the use of layers (see 5.7.1) in order to prevent them from being rendered by the main camera as well. The lines, as well as the colored squares, then are drawn by directly using OpenGL functions, which are executed in the *OnGUI* method of the GUIObj GameObject.

For each line (or incoming signal), an array exists which is capable of holding 300 values, and is read out continuously for rendering the lines. After 300 values have been received and the array is full, an index value is used for determining the start of the line as well as the position for storing the next value in the array. This index value is increased each time, a value is received, and resetted if exceeding the capacity of the array. The same process happens for all lines (or incoming signals) used.

### 5.7.3 DataLogger

The DataLogger component is attached to the client, and remotely controlled via RPC functions which are received from the server. The reason for this setup is to be capable of recording the situation of the simulation as it occurs in the client, and not as it is forwarded (in a limited way) to the server.

As no knowledge had been available with regards to the requirements of collecting data for later analysis, the decision was made to record the position and orientation of the virtual camera (head) and the arm, and additionally the values of the eight EMG controlling signals. Later the threshold value, which is used for deciding, whether an object was grasped or not, was added to the data log. More information about the practicability of these recordings is presented in chapter 6.1.3.

CHAPTER 6

# Results

Since the scope of the application already has been presented in the previous chapters, in this chapter no summary of the final resulting software will be given.

After the implementation of the second part of the work was finished, the capability of the application was tested by creating several scenarios, which were performed by eight probands. The four basic scenarios of these tests as well as the reactions of the probands during and after performing the exercises will be presented in the first section of this chapter. Furthermore, the attempt of analyzing the data, which was recorded during these tests by using the data logger, will be presented.

In the second section, a general discussion about the implementation of the work is presented. Furthermore, the remaining problem of capturing data for analysis is treated separately, as well as the difficulties, experienced with the primal grasping approach.

Due to the experiences made during the design and implementation phase of the work, but especially due to the experiences made while user-testing and according to the feedback of the probands, in the third section the design of the interaction interface will be discussed, especially with regards to the design of the environment and the creation of exercise scenarios.

## 6.1 User Tests

At the end of the project, the virtual reality system was tested in practice with eight probands, four healthy persons and four forearm amputees. In order to perform these tests, four different scenarios where created with the idea of increasing the difficulty by each scenario. Additionally, each of these scenarios had to be performed several time with different mappings by each of the probands.

In the first subsection, the creation of the four basic scenarios will be described in detail and thereby gives an example of using the virtual reality system in a practical manner. In the second subsection, the reactions of the probands is presented and discussed, with regards to the virtual reality experience on the one hand, and the grasping process in particular on the other hand. As

the intention of performing these user tests partially was to record the processes of performing exercises for later analyzing and evaluating, in the third subsection an overview of this difficulty is given.

### 6.1.1 Scenarios

The scenarios created for the user tests basically consist of three different grasping tasks, which had to be performed up to eight times respectively with a different hand mapping setup and using different training objects (with regards to their grip force interval). Additionally, an fourth task was created, which only makes use the EMG graph control window of the sever application for practicing the creation of EMG signals. In the following paragraphs, each of these scenarios will be described in detail. To give an idea of the mapping setups used with these scenarios, for example, instead of using two signals for opening and closing the hand, only one signal was used, which again was calculated by the EMG tracking device, according to the two signals created by the proband. In another setup, the EMG tracking device was sending the data to a real Michelangelo Hand prosthesis, which then performed the grasping action, while forwarding the position of the hand as well as the grip force to the simulation. For all the scenarios and their variations, only two hand states were used, one for an opened, and one for a closed hand. These states were created with regards to the states of the real Michelangelo Hand prosthesis.

#### The MyoTrainer

This test scenario was not intended while implementing the application, and does not require the virtual reality system itself. In this scenario, the proband is wearing the EMG tracking device and is placed in front of a monitor, which displays the EMG Graph control window of the server application. In this window, as already mentioned, all incoming EMG control signals are visualized in different colors. The task for the probands was to create EMG controlling signals, respectively by one electrode, which have to follow a sinus curve signal, created by the EMG tracking device and therefore also displayed in the EMG graph (see fig.6.1). The configuration and activation of this sinus curve was done by embedded commands. For each proband, three different tests have been performed, using three different frequencies (20, 50 and 80 Hz).

#### Simple Grasping Interaction

For introducing the proband into the virtual grasping interaction, this scenario makes use of the Permanent Object mode, as it is described in chapter 4.7.3. The proband is sitting but already wearing a HMD and looking towards the (virtual) arm. As the training object to be grasped is attached to the hand, and therefore will follow all movements of the arm as well, the proband only has to concentrate on the process of closing the hand without exceeding the required grip force. The arm does not have to be moved at all. As well as the following scenarios, this grasping scenario was performed twice for each mapping setup with different grip force intervals for the training objects.

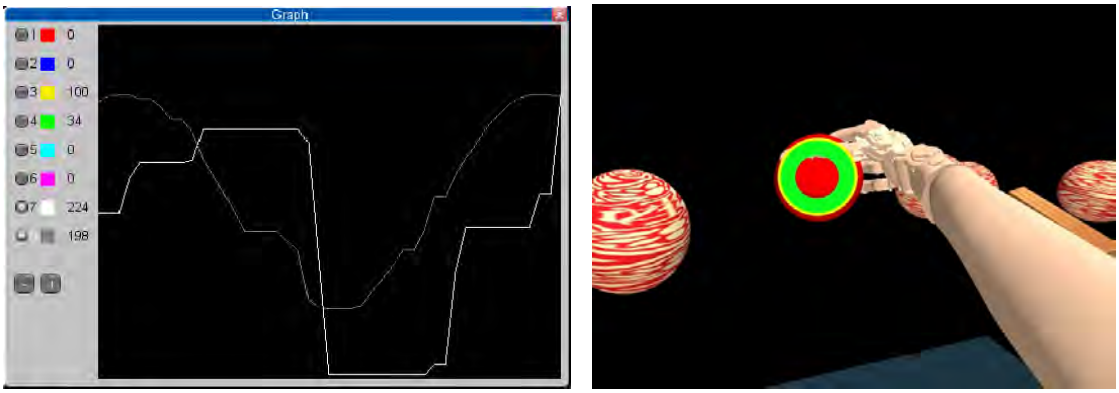



**Figure 6.1:** Left: The EMG graph used as a Myo-Trainer. The task was to follow the gray line with the white line by creating the appropriate EMG signals. The edges in the displayed sinus curve are caused by the irregularly intervalic synchronization process between client and server. Right: The *Grasping and Depositing in Motion* scenario, from the viewpoint of the protagonist. The completely red ball is held and moved by the proband.

### Statical Grasping and Depositing

In order to slightly increase the difficulty of interacting, in this scenario the proband is still sitting, but in contrast to the previous scenario, the training object is not mounted to the hand but has to be grasped and deposited each time.

This scenario was achieved by saving a training object scenario, which only contains one object. At the beginning of the exercise the scenario is loaded, and the proband has to be placed in front of the object. The elbow of the proband is placed on a table and only the arm is moved. For performing the task, the proband has to grasp the object, located in the middle of the field of vision, with the appropriate grip force and move it towards the left or right before depositing it again. After depositing, the exercise instructor has to reload the training object scenario, causing the object to be moved back to its primal position. As this resetting can be done instantly (if the exercise instructor is attentive), fluent grasping and depositing actions can be performed by the proband.

### Grasping and Depositing in Motion

Finally, in this fourth scenario the proband is moving around freely. The task of this scenario is to grasp and deposit 32 training objects, which are placed as a circle with a diameter of two meters in the middle of the tracking area (the area of the tracking area is 4x4 meters). For depositing, primarily the depositing area was intended, but due to the main goal of testing and observing the grasping interaction, it was decided to be sufficient, if the protagonist is just grasping an object with the appropriate grip force (which is the actual exercise) and moving it slightly to the side before releasing it (see fig. 6.1).

This scenario also uses training object scenarios, which in contrast to the previous scenario presented, are only loaded once. The scenario remains, until the propand finished grasping all

objects in the circle. For positioning the training objects for matching a circle, the coordinates have been calculated manually and the training object file was created in a text editor (see 6.2).

**Figure 6.2:** The *Grasping in Motion* scenario. The task of this scenario was to grasp and slightly move away the training objects before releasing them again. The objects were aligend in the shape of a circle, placed in the middle of the interaction (tracking) area.

### 6.1.2 User Feedback

All the probands were able to manage the interaction of grasping within minutes, even the hardest task of grasping and depositing in motion. After the exercises were performed, three of the four amputees replied, that the feeling of the grasping process changed from creating the appropriate EMG signals to "directly" interacting with the virtual hand. In relation to the age of the probands, after two to three hours first signs of fatigue arised, especially in the muscles which were used for creating the EMG signals. The virtual reality experience itself was evaluated positively by all the probands. All probands replied, that the movements they performed did match the image in virtual reality. Furthermore, seven of the eight probands replied that moving the virtual arm felt like moving their own arm.

Problems were caused by the black background of the environment, which partially led to problems of orientation due to the uniform coloring, especially at the beginning of the tests.

One of the probands started to feel sick after about three hours of testing (breaks included), which finally required to break the tests for this proband. According to himself, the sickness was primarily caused by the action of looking down and up again. This can be due to fatigue caused by looking into the HMD, which gets demanding for the eyes after a while. Furthermore, it can also be due to the uniform background, which next to orientation problems created a strong contrast to the bright colorful objects in the foreground, causing additional stress for the eyes.

Finally, the probands were asked to rate their favorite test scenario out of the four, presented in the previous subsection. Most of the probands claimed, that the third and especially the fourth scenario was the best, which is the scenario with the highest level of difficulty. One proband even stated, that these scenarios were better because they are more meaningful. Furthermore, it

was noticeable that the repetition of the four scenarios with several mapping modes decreased the motivation of the probands and their effort to perform the exercise without mistakes.

### 6.1.3 Data Evaluation

For analyzing the movements of the probands, for example, with regards to different simulations of grip force, or for illustrating the improvement of the grasping process, a data logger was implemented (see chapter 5.7.3). This functionality records the position and orientation of the protagonist's head as well as of the arm target (which approximately also defines the position and orientation of the hand). Additionally, all incoming EMG signals, and the hand object capture threshold for determining the closing position of the hand.

Originally, the idea behind this selection of data was to implement a "player" function for replaying the recorded actions for further analysis. Unfortunately, due to time restrictions it was not possible to implement such a functionality. The approach of directly analyzing the recorded data resulted in the problem, that the particular data sets were not recorded at regular intervals (about 24 times per second). Since no timestamp was created with these data sets, it was not possible to retrace the action properly.

Due to this difficulty, but furthermore due to the problem, that no specific thoughts were made about how to analyze these data before recording it, afterwards it turned out to nearly be impossible reusing this data for any interpretations. In the following chapter 6.2, a short discussion about possible and useful approaches of recording data is given.

## 6.2 Implementation

The implementation of the work was mainly done in Unity3D for creating the server and the client application. Basically, implementing the virtual reality environment was easy and not technically challenging, as Unity3D takes care of performance issues and provides a scalable 3D engine with a huge scope of features. Furthermore, next to the unreachable quality of such a professional and therefore almost bug-free engine, Unity3D also provides support for several platforms like Windows and MacOSx, which are available in the free version.

Generally, the most difficulties while implementing were caused by additional requirements arising, whose implementations caused conflicts with the already implemented architecture. This especially happened during the second part of the work, while designing the functionalities for creating various testing scenarios. This resulted, for example, in the unusual approach for saving the options for simulating grip force in a Speed Mode or Position Mode mapping (as described in chapter 5.5.4).

A major difficulty was the implementation of the primal, physics engine supported grasping interaction. Since the difficulty, as it appeared during the project, was already discussed in chapter 5.6.1, in the following subsection a possible approach how to overcome these problems will be presented, which unfortunately originated after finishing this project.

The second unsolved issue with regards to the implementation is the data logger. In the previous section 6.1, the difficulty of using the data, which was recorded during the user-tests,

is presented. In the secondly following subsection, more thoughts will be made with regards to providing useful data for later analyzing and evaluating.

### 6.2.1 Physx Powered Grasping

The grip force sensitive, new grasping process as implemented in the second part of the work, not only has proven to be more than sufficient for providing an easy to learn and immersive grasping interaction. Furthermore, the user-tests performed have clearly shown, that it is not necessary to provide a super-realistic grasping process to achieve the goal of creating an immversive experience. Nevertheless, in the following an approach is presented, which might help to overcome the problems of unwanted physical behavior, as they are described in chapter 5.6.1.

As stated in the reference of the built-in Physx engine, using a wrong-scaled scenery can cause problems when using Rigidbody components. In general, for providing a working physics simulation, it is suggested to use scales, which are according to the real world. For example, the size of a person should be about 1.5-2 units (meters).

Basically, this was fulfilled when designing the applications, not least because the position data, received by the ioTracker system also is measured in meters. The reason for the still occurring problems might be caused due to the fact, that in usual gaming scenarios, as Unity3D is made for, the size of objects commonly affected by the physics simulation is not smaller than a box or a cup. The relative small size of the fingers, in comparison with these objects, may be too small in order to provide a proper behavior of the physics engine of Unity3D.

This insight emerged during the work on a slightly modified application which was also based on the virtual hand model, as it is used for this work. Since it was required to use the physics simulation for creating a virtual hand, capable of pushing away objects it collides with, similar problems were experienced as during the implementation of the primal grasping interaction. Especially, it was not possible to detect the collision between a particular finger and another object properly.

Out of necessity, the try was made to increase the scale of the whole environment by the factor 10, which immediately led to the success of solving this particular problem. As the physics simulation, which processes in this other project is not as complex as the collisions, which are occuring when grasping an object with a hand (as it was implemented in the primal grasping approach), the attempt still has to be made, if this solution would also improve this particular problem. On the other hand, as the user-tests have shown, such a realistic approach might not provide a better experience than the one, finally used for this work, is capable of.

### 6.2.2 Data Logger

As mentioned in the previous section, one problem when using the recorded data was the absence of a timestamp. Such a timestamp could have been used for retracing the intervals between the particular data sets. Indeed, if the existence of such a timestamp would have helped in evaluating the data is to be seriously doubted. Such has the recording of the probands head and arm position proven to be completely useless for evaluating the action of moving towards an object. This matter could maybe be better evaluated, for example, by measuring the distance of the hand to the nearest object, or to the object the hand is pointing towards. In a similar way it

would be necessary to create several more indicative values, like the already implemented hand-object-capture threshold value, for better describing the grasping interaction than by recording the incoming EMG controlling signals.

The problem of measuring in accurate intervals can be overcome by using a timestamp for each data set recorded. Providing data, which is recorded in (small) regular intervals is not possible, as the *update* method of an GameObject as the lowest level of the implementation structure in Unity3D is not called regularly either. Therefore the recording intervals have to be chosen smaller than the required temporal resolution.

## 6.3 Discussion about the Design

In this chapter, the design of the interaction interface, including the hardware used as well as the grasping interaction itself and finally the virtual environment setting as a whole will be discussed. Experiences regarding this issue were acquired during the implementation process, but not surprisingly especially when performing the user-tests.

### 6.3.1 Hardware

As presented in chapter 4.1, the protagonist is wirelessly connected to the immobile parts of the virtual reality system. While this setup basically has proved its benefits, the Bluetooth connection to the EMG tracking device did break down several times during the tests for each proband. It was not possible to find any rational reason for this problem.

The first HMD used in this work was an *emagin Z800* and had a resolution of 800x600 pixels. This has proven to be quite exhausting for the eye, when wearing the display for a longer time. The second HMD used, the Silicon Micro Display ST1080, had a resolution of 1920x1080 pixels (Full HD). This significantly increased the comfort of looking into the display. Unfortunately, this second HMD was shaped in a way that it absolutely did not match the shape of a human head, which resulted in a barely more pleasant virtual reality experience.

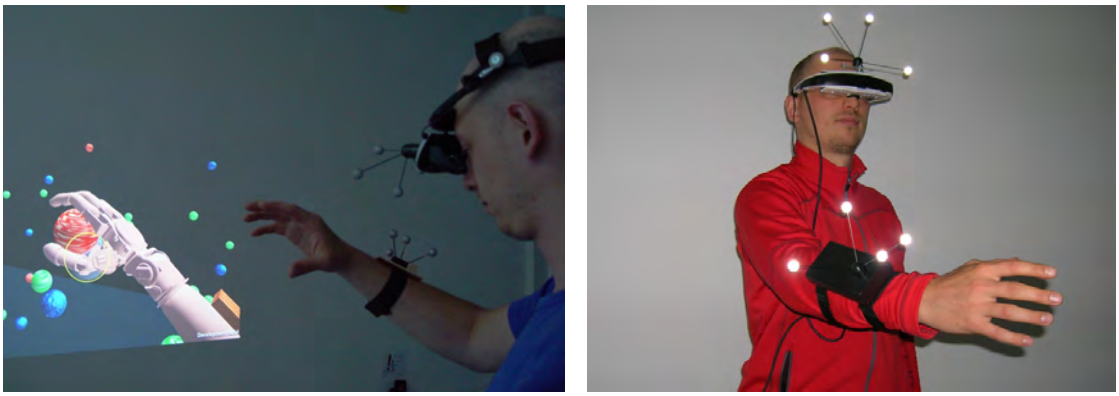


**Figure 6.3:** Left: The emagin Z800 HMD. Right: The Silicon Micro Display ST1080 HMD.

### 6.3.2 Grasping Interaction & Aides

The implementation of the new grasping process in comparison to the first one has the following benefit for the protagonist: When moving towards an object, the protagonist does not have to care about not touching the object before holding it, since it does not interact physically with the hand at all. As the test runs have shown, moving the hand towards the right position aided by the hand object shadow was easy to learn for the probands. This allows a protagonist to concentrate on the real task of closing the hand and creating the appropriate amount of grip force.

The usefulness of the hand object shadow has also been proven due to a certain bug, which appeared during the user tests. When reloading a training object scenario, the objects were always created with the default size, even if the size of the objects had been adjusted manually. This bug resulted in training objects being smaller than the hand object shadow, which was interpreted by the probands with regards to the perspective view, as the hand object shadow being too far away when grasping an object and led them to always grasp too far behind the object. This illustrates the effectiveness of the grasping aide for finding the right position of the training object, and with regards to the feedback of the probands it does not influence the realism of the grasping process.

Another bug, resulting from the difficulty of mirroring the whole environment for creating a left-handed virtual arm and hand (see chapter 5.3.2), clearly shows that the exactness of the visualization process is not essential for creating an immersive user experience.

For still being able to perform the tests with a left-handed proband, the arm target was attached in a way that the right hand was positioned as left hand, with the thumb pointing downwards. The proband equipped with this setup did neither significantly needed more time to learn the grasping interaction, nor did he experienced more difficulties when performing the exercises. The use of the hand object shadow as well as the approximate visualization of closing and opening of the hand obviously are sufficient for a proper grasping interaction. Therefore, such a "realistic" visualization of a grasping process, as it was primarily intended, is not necessary at all for creating an immersive virtual reality experience for the protagonist while exercising. And this, after all, was the original intention behind creating a realistic grasping process.

### 6.3.3 Training Environment

Similar to the grasping interaction, the approach of designing the virtual environment has fundamentally changed from a playful and friendly environment to a clean and minimalist one. The idea behind this was to support the exercising task by not distracting the protagonist from the interaction.

According to the feedback given by the probands, neither the danger of being distracted by the environment was given, nor did the minimalist design, especially the black background, support them in any way. A few proband had problems with orienting in the virtual environment due to the uniform black coloring of the background. Furthermore, due to the obvious decreasing of the probands motivation while repeating the same scenarios with different mapping settings, it can be assumed that an approach similar to the first design of the environment would be more

supporting for the user after all. For providing more variety, the environment as well as the tasks have to differ more for keeping the motivation of the probands at a high level.

With regards to serious games, such an environment does not have to be detailed or complex at all. Therefore it would be easy to create such environments in large numbers. Serious game scenarios, such as catching oranges from a tree (presented in [3]), create a meaningful context and therefore provide the user in finding a challenge, like catching as many oranges as possible in a certain time. Such a challenge is harder to find, if the context of the tasks, as in this work, is more or less without any meaning.

CHAPTER 7

# Summary and Future Work

The application presented in this thesis provides a full immersive virtual reality experience for performing exercises with a virtual hand (prosthesis). This experience is created by the interaction of the ioTracker tracking system, the EMG tracking hardware provided by Otto Bock, the OpenTracker framework and finally the game engine Unity3D.

The motion of the protagonist's head and arm are tracked with the visual marker-based tracking system ioTracker, and then forwarded through the OpenTracker framework as well as the EMG controlling signals for the virtual hand. These signals again are originally created as EMG signals by the protagonist and then processed and transmitted by the EMG tracking device provided by Otto Bock. Using the OpenTracker framework as an interface for receiving and sending (tracking) data allows for easy use of different technologies for tracking the motions of the protagonist as well as for creating EMG controlling signals without any need for re-implementing parts of the Unity3D applications. The OpenTracker framework forwards the received tracking data to a Unity3D application, which is the client, creating the visual output for the head mounted display (HMD) the protagonist is wears.

For interacting in the virtual environment, the protagonist simply has to move his arm (stump) with the tracking target attached, in order to stear the virtual arm and hand towards a virtual object. For the grasping motion itself, the appropriate EMG controlling signals are created by contracting and relaxing the respective muscles. The environment itself only consists of a black background and a green floor with the approximate dimensions of the tracking volume of about 4x4 meters. In the space above this accessible area, the sphere shaped training objects are placed freely in the air. Several optical aides support the protagonist during the grasping action, first by moving the hand correctly over an object, and secondly by creating the appropriate amount of grip force for grasping and holding the particular object.

A second Unity3D application, the server, functions as the controlling unit. This application is connected to the client via a network, and allows to adjust particular parts of the simulation process. The behavior of the virtual hand, respectively prosthesis, can thus be defined according to the incoming EMG signals created by the protagonist. Additionally, the simulation of grip force can be activated or even calculated outside of the application. A hand state editor allows

to define any type of hand position for later use in the controlling mechanism for the virtual hand. The virtual arm the hand is attached to can be adjusted in size and shape for best matching the real circumstances. Furthermore, it is possible to dynamically specify the position of the tracking target, which is mounted to the protagonist's arm (stump). This setup allows to locate the virtual hand at part in space, the protagonist would expect his real hand to be.

For additional configuration of the EMG tracking device, the functionality of embedded commands, as provided in the sever application, allows to easily send control commands to the device while receiving the EMG control signal data as created by the protagonist. To achieve this, the interface to the OpenTracker framework had to be extended by this functionality as well.

Finally, the server application provides the possibility of creating and performing exercising scenarios, which is the actual main task with regards to the creation of a virtual evaluation and training environment. This functionality makes it possible to save and reload a certain adjustment of training objects in the environment. For each training object the particular grip force interval can be specified. In addition to the scenario of just grasping and releasing these objects, the exercising task can be extended by using a depositing area, either positioned in the environment or virtually attached to the protagonist. A second test scenario is provided under the name "Permanent Object Mode", and allows to practice only the grasping interaction without heading towards a training object, by providing an object attached to the virtual hand. For creating even more basic test scenarios, the EMG signal graph can be used for concentrating only on creating the appropriate EMG signals.

### Future Work

As the user-tests, which were performed at the end of the work (see chapter 6.1) have shown, the virtual grasping interaction as it is presented in this thesis is sufficient for creating an immersive training experience. Based on this idea, the application could be extended by the option of using different environments for creating a more varied offer of possible exercise scenarios. As the action to be exercised cannot provide much diversity in the exercising progress, this is a possible alternative.

The goal of such an approach would be to create a gaming experience for the patient, which helps him keeping the level of motivation during the whole process of rehabilitation. Besides the diversity, provided by alternating environments and exercising task, another important aspect for creating a gaming experience is the use of a scoring system. For example, the grasping process can become more challenching, if the task is given to grasp as many objects as possible in a limited amount of time. Using scores can also be a simple and motivating way of illustrating the progress of the rehabiliation process to the patient.

In addition, the application presented in this thesis could also be used, for example, for post-stroke rehabilitation regarding upper limbs. The use of EMG signals, at it is used in this work for controlling the virtual hand, is a common approach in works regarding the issue of upper limb post-stroke rehabilitation (see chapter 2).

# Bibliography


[1] Adel Al-Jumaily and Ricardo A Olivares. Electromyogram (emg) driven system based virtual reality for prosthetic and rehabilitation devices. In *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*, pages 582–586. ACM, 2009.

[2] Grigore Burdea, Viorel Popescu, Vincent Hentz, and Kerri Colbert. Virtual reality-based orthopedic telerehabilitation. *Rehabilitation Engineering, IEEE Transactions on*, 8(3):430–432, 2000.

[3] James William Burke, MDJ McNeill, DK Charles, Philip J Morrow, JH Crosbie, and SM McDonough. Serious games for upper limb rehabilitation following stroke. In *Games and Virtual Worlds for Serious Applications, 2009. VS-GAMES'09. Conference in*, pages 103–110. IEEE, 2009.

[4] James William Burke, MDJ McNeill, DK Charles, Philip J Morrow, JH Crosbie, and SM McDonough. Augmented reality games for upper-limb stroke rehabilitation. In *Games and Virtual Worlds for Serious Applications (VS-GAMES), 2010 Second International Conference on*, pages 75–78. IEEE, 2010.

[5] Yang-Wai Chow. Low-cost multiple degrees-of-freedom optical tracking for 3d interaction in head-mounted display virtual reality. *ACEEE International Journal on Network Security*, 1(1), 2010.

[6] Jaeyong Chung, Namgyu Kim, Jounghyun Kim, and Chan-Mo Park. Postrack: a low cost real-time motion tracking system for vr application. In *Virtual Systems and Multimedia, 2001. Proceedings. Seventh International Conference on*, pages 383–392. IEEE, 2001.

[7] Mathis Csisinko and Hannes Kaufmann. Cutting the cord: Wireless mixed reality displays. In *Proceedings of the Virtual Reality International Conference (VRIC 2011)*, 2011. Vortrag: Laval Virtual - Virtual Reality International Conference 2011, Laval, France; 2011-04-06 – 2011-04-08.

[8] D Shefer Eini, N Ratzon, AA Rizzo, SC Yeh, B Lange, B Yaffe, A Daich, PL Weiss, and R Kizony. A simple camera tracking virtual reality system for evaluation of wrist range of motion. 2010.

[9] Eletha Flores, Gabriel Tobon, Ettore Cavallaro, Francesca I Cavallaro, Joel C Perry, and Thierry Keller. Improving patient motivation in game development for motor deficit rehabilitation. In *Proceedings of the 2008 International Conference on Advances in Computer Entertainment Technology*, pages 381–384. ACM, 2008.

[10] Anders Fougner, Erik Scheme, Adrian DC Chan, Kevin Englehart, and Øyvind Stavdahl. Resolving the limb position effect in myoelectric pattern recognition. *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, 19(6):644–651, 2011.

[11] Rhona Guberek, Sheila Schneiberg, Patricia McKinley, Felicia Cosentino, Mindy F Levin, and Heidi Sveistrup. Virtual reality as adjunctive therapy for upper limb rehabilitation in cerebral palsy. In *Virtual Rehabilitation International Conference, 2009*, pages 219–219. IEEE, 2009.

[12] Markus Hauschild, Rahman Davoodi, and Gerald E Loeb. A virtual reality environment for designing and fitting neural prosthetic limbs. *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, 15(1):9–15, 2007.

[13] S Herle, S Man, Gh Lazea, C Marcu, P Raica, and R Robotin. Hierarchical myoelectric control of a human upper limb prosthesis. In *Robotics in Alpe-Adria-Danube Region (RAAD), 2010 IEEE 19th International Workshop on*, pages 55–60. IEEE, 2010.

[14] XL Hu, KY Tong, XJ Wei, W Rong, EA Susanto, and SK Ho. The effects of post-stroke upper-limb training with an electromyography (emg)-driven hand robot. *Journal of Electromyography and Kinesiology*, 2013.

[15] iotracker Tracking System. http://www.iotracker.com. Accessed: 2013-2-20.

[16] Abhishek Kar. Skeletal tracking using microsoft kinect. *Methodology*, 1:1–11, 2010.

[17] Takehito Kikuchi, Hu Xinghao, Kazuki Fukushima, Kunihiko Oda, Junji Furusho, and Akio Inoue. Quasi-3-dof rehabilitation system for upper limbs: Its force-feedback mechanism and software for rehabilitation. In *Rehabilitation Robotics, 2007. ICORR 2007. IEEE 10th International Conference on*, pages 24–27. IEEE, 2007.

[18] E Lamounier, Kenedy Lopes, Alexandre Cardoso, Adriano Andrade, and Alcimar Soares. On the use of virtual and augmented reality for upper limb prostheses training and simulation. In *Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE*, pages 2451–2454. IEEE, 2010.

[19] BA Lock, K Englehart, and B Hudgins. Real-time myoelectric control in a virtual environment to relate usability vs. accuracy. Myoelectric Symposium, 2005.

[20] Xun Luo, Robert V Kenyon, Tiffany Kline, Heidi C Waldinger, and Derek G Kamper. An augmented reality training environment for post-stroke finger extension rehabilitation. In *Rehabilitation Robotics, 2005. ICORR 2005. 9th International Conference on*, pages 329–332. IEEE, 2005.

[21] Annette Mossel, Christian Schönauer, Georg Gerstweiler, and Hannes Kaufmann. Artifice - augmented reality framework for distributed collaboration. *The International Journal of Virtual Reality*, 11(3):1–7, 2012.

[22] J Rafiee, MA Rafiee, F Yavari, and MP Schoen. Feature extraction of forearm emg signals for prosthetics. *Expert Systems with Applications*, 38(4):4058–4067, 2011.

[23] F Sebelius, M Axelsson, N Danielsen, J Schouenborg, and T Laurell. Real-time control of a virtual hand. *Technology and Disability*, 17(3):131–141, 2005.

[24] Takaaki Shiratori, Hyun Soo Park, Leonid Sigal, Yaser Sheikh, and Jessica K Hodgins. Motion capture from body-mounted cameras. In *ACM Transactions on Graphics (TOG)*, volume 30, page 31. ACM, 2011.

[25] Cara E Stepp, James T Heaton, Rebecca G Rolland, and Robert E Hillman. Neck and face surface electromyography for prosthetic voice control after total laryngectomy. *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, 17(2):146–155, 2009.

[26] Studierstube OpenTracker Framework. http://studierstube.icg.tugraz.at/opentracker/. Accessed: 2013-4-20.

[27] Toyokazu Takeuchi, Takahiro Wada, Masato Mukobaru, et al. A training system for myoelectric prosthetic hand in virtual environment. In *Complex Medical Engineering, 2007. CME 2007. IEEE/ICME International Conference on*, pages 1351–1356. IEEE, 2007.

[28] Unity3D Game Engine. http://www.unity3d.com. Accessed: 2013-2-26.

[29] Unity3D Scripting Reference. http://docs.unity3d.com/documentation/scriptreference/index.html. Accessed: 2013-3-25.

[30] Robert Y Wang and Jovan Popović. Real-time hand-tracking with a color glove. In *ACM Transactions on Graphics (TOG)*, volume 28, page 63. ACM, 2009.

[31] Huiyu Zhou and Huosheng Hu. Human motion tracking for rehabilitation—a survey. *Biomedical Signal Processing and Control*, 3(1):1–18, 2008.